Contents

Data Compression
Lecture Notes

Dr. Faisal Aslam

1: Lecture 1: Introduction to Data Compression

1.1 Learning Objectives

By the end of this lecture, students will be able to:

- Understand the motivation and benefits of data compression
- Differentiate between lossless and lossy compression techniques
- Compute and interpret common compression performance metrics
- Apply the Huffman coding algorithm step by step
- Analyze real-world compression trade-offs

1.2 Introduction and Motivation: Why Compress Data?

Data compression is the process of representing information using fewer bits than its original form. It is a fundamental component of modern computing systems, enabling efficient storage, faster communication, and reduced operational costs.

Everyday applications of compression include:

- Streaming audio and video
- Image storage and sharing
- File archiving and backups
- Network communication and cloud services

Definition

Data Compression is the process of reducing the number of bits required to represent information, either:

- **Losslessly**: allowing exact reconstruction of the original data
- **Lossily**: allowing controlled loss of information to achieve higher compression

1.2.1 Benefits of Data Compression

Data compression provides three key benefits that are critical in modern computing:

1. **Reduce Storage Space**:
   - Allows more data to be stored in the same physical space
   - Enables archival of historical data that would otherwise be discarded
   - Reduces hardware requirements for storage systems
2. **Reduce Communication Time and Bandwidth**:
   - Enables faster file transfers and downloads
   - Makes high-quality streaming (4K/8K video) practical over limited bandwidth
   - Reduces latency in real-time applications like video conferencing and online gaming
   - Allows IoT devices to transmit data efficiently over wireless networks
3. **Save Money**:
   - Reduces cloud hosting costs (storage and egress fees)
   - Lowers communication costs for data transmission
   - Decreases capital expenditure on storage hardware
   - Reduces energy consumption for data centers and network infrastructure

1.3 Lossless vs. Lossy Compression

1.3.1 Lossless Compression

Lossless compression guarantees perfect reconstruction of the original data. It is essential when accuracy and data integrity are critical.

Typical applications:

- Text files and source code
- Executables and databases
- Medical, scientific, and legal data

1.3.2 Lossy Compression

Lossy compression achieves higher compression ratios by discarding information that is less perceptible or less important.

Typical applications:

- Audio (MP3, AAC)
- Images (JPEG)
- Video (H.264, H.265)

1.3.3 Choosing Between Lossless and Lossy

Factor	**Lossless Compression**	**Lossy Compression**
Reconstruction	Exact	Approximate
Data sensitivity	High	Moderate to low
Typical ratios	Low to moderate	High
Quality impact	None	Controlled degradation

Table 1: Lossless vs. Lossy Compression

1.4 Compression Performance Metrics

1.4.1 Size-Based Metrics

$$\begin{aligned}
\text{Compression Ratio (CR)} &= \frac{\text{Original Size}}{\text{Compressed Size}} \\
\text{Compression Factor} &= \frac{\text{Compressed Size}}{\text{Original Size}} \\
\text{Space Savings (\%)} &= \left(1 - \frac{\text{Compressed Size}}{\text{Original Size}}\right) \times 100\%
\end{aligned}$$

Interpretation:

- Larger compression ratios indicate better compression
- Smaller compression factors indicate better compression

1.4.2 Rate-Based Metrics

$$\begin{aligned}
\text{Bits per Sample (bps)} &= \frac{\text{Compressed Size (bits)}}{\text{Number of samples}} \\
\text{Bit-rate (bps)} &= \frac{\text{Compressed Size (bits)}}{\text{Time (seconds)}}
\end{aligned}$$

These metrics are particularly important in audio and video compression systems.

1.5 Worked Example: Audio Compression Metrics

Example

Uncompressed Audio Properties

- Duration: 180 seconds
- Sampling rate: 44.1 kHz
- Bit depth: 16 bits
- Channels: 2 (stereo)

Original Size Calculation

$$\begin{aligned}\text{Total samples} &= 180 \times 44{,}100 \times 2 = 15{,}876{,}000 \\ \text{Size (bits)} &= 15{,}876{,}000 \times 16 = 254{,}016{,}000 \\ \text{Size (MB)} &= \frac{254{,}016{,}000}{8 \times 1{,}048{,}576} \approx 30.27\end{aligned}$$

Compression Results

Method	Size (MB)	CR	Savings	Bit-rate
FLAC (lossless)	18.16	1.67:1	40%	807 kbps
MP3 @ 320 kbps	6.75	4.49:1	77.7%	320 kbps
AAC @ 256 kbps	5.40	5.61:1	82.2%	256 kbps

1.6 Huffman Coding

Huffman coding is a widely used **lossless compression algorithm** that assigns variable-length binary codes to symbols based on their frequencies. More frequent symbols receive shorter codes.

1.6.1 Step-by-Step Huffman Coding Example

Example

Message: `MISSISSIPPI RIVER` (17 characters including space)

Symbol Frequencies

Symbol	Frequency
I	5
S	4
P	2
R	2
M	1
V	1
E	1
(space)	1

Tree Construction

1. Combine M(1) + V(1) → 2
2. Combine E(1) + (space)(1) → 2
3. Combine P(2) + R(2) → 4
4. Combine 2 + 2 → 4
5. Combine 4 + 4 → 8
6. Combine I(5) + S(4) → 9
7. Combine 8 + 9 → 17

One Possible Code Assignment

Symbol	Code	Length
I	00	2
S	01	2
P	100	3
R	101	3
M	1100	4
V	1101	4
E	1110	4
(space)	1111	4

Compressed Size

$$5(2) + 4(2) + 2(3) + 2(3) + 4(1) = 52 \text{ bits}$$

Original Size (ASCII) = $17 \times 8 = 136$ bits
Compression Ratio = $136/52 \approx 2.62 : 1$

1.6.2 Key Properties of Huffman Coding

- Produces prefix-free codes
- Enables instantaneous decoding
- Guarantees minimum average code length among prefix codes
- Widely used in practical compression systems

1.7 End of Chapter Questions

Exercise 1.0

Problem 1: Basic Compression Metrics

An uncompressed grayscale image has the following properties:

- Resolution: 1024×1024 pixels
- Bit depth: 8 bits per pixel

After compression, the image size is 320 KB.
Calculate:

(a) Original image size in KB

(b) Compression ratio

(c) Compression factor

(d) Space savings percentage

Exercise 1.1

Problem 2: Audio Bit-rate and Storage

A mono audio recording has the following parameters:

- Duration: 5 minutes
- Sampling rate: 48 kHz
- Bit depth: 16 bits

The file is compressed using a lossy codec to a constant bit-rate of 192 kbps.
Calculate:

(a) Size of the uncompressed audio file in MB

(b) Size of the compressed file in MB

(c) Compression ratio

(d) Bits per sample after compression

Exercise 1.2

Problem 3: Comparing Compression Options

A video clip has an uncompressed data rate of 120 Mbps. Three compression options are available:

Option	Compressed Bit-rate
A	6 Mbps
B	3 Mbps
C	1.5 Mbps

For each option, calculate:

(a) Compression ratio

(b) Data consumed for a 10-minute video (in MB)

Which option would you choose for:

(i) Live video streaming?

(ii) Archival storage?

Briefly justify your answers.

Exercise 1.3

Problem 4: Huffman Coding Construction

Given the following symbol frequencies:

Symbol	Frequency
A	10
B	8
C	6
D	5
E	4
F	3
G	2
H	2

(a) Construct the Huffman tree step by step

(b) Assign a binary code to each symbol

(c) Compute the total number of bits required to encode the message

(d) Calculate the average number of bits per symbol

Exercise 1.4

Problem 5: Fixed-Length vs. Huffman Coding
Using the symbol set from Problem 4:

(a) Determine the minimum fixed-length code required

(b) Compute the total number of bits using fixed-length coding

(c) Compare the result with Huffman coding

(d) Calculate the percentage reduction in total bits achieved by Huffman coding

Exercise 1.5

Problem 6: Text Compression Scenario
A text file contains 50,000 characters and is stored using 8-bit ASCII encoding. After compression using a lossless algorithm, the file size becomes 18 KB. Calculate:

(a) Original file size in KB

(b) Compression ratio

(c) Compression factor

(d) Space savings percentage

Explain why compression ratios for text files vary significantly depending on content.

Exercise 1.6

Problem 7: Practical Design Question
You are designing a compression system for a wearable health-monitoring device that:

- Records sensor data continuously
- Has limited storage capacity
- Requires exact data reconstruction
- Operates on a low-power processor

(a) Should the system use lossless or lossy compression? Explain.

(b) Which performance metrics are most important in this scenario?

(c) Would a variable-length coding scheme be appropriate? Why or why not?

2: Lecture 2: Theory of Compression — Limits and Optimality

2.1 Types of Codes: From Ambiguous to Instantaneous

Definition

Types of Codes

- **Non-singular Code**: Each source symbol maps to a distinct codeword

$$x_i \neq x_j \Rightarrow C(x_i) \neq C(x_j)$$

- **Uniquely Decodable Code**: Every finite sequence of codewords corresponds to exactly one sequence of source symbols

$$C(x_1)C(x_2)\cdots C(x_n) = C(y_1)C(y_2)\cdots C(y_m) \Rightarrow n = m \text{ and } x_i = y_i$$

- **Prefix Code (Instantaneous Code)**: No codeword is a prefix of another codeword

$$\forall i \neq j : \quad C(x_i) \text{ is not a prefix of } C(x_j)$$

Key Relationships:

$$\text{Prefix Codes} \subset \text{Uniquely Decodable Codes} \subset \text{Non-singular Codes}$$

Important

Why Prefix Codes are Special

- **Instantaneous decoding**: Can decode as soon as codeword ends (no lookahead needed)
- **Tree representation**: Always correspond to leaves of a binary tree
- **Kraft inequality**: Always satisfy $\sum 2^{-\ell_i} \leq 1$
- **Practical**: Used in Huffman coding, many real-world compressors

Example

Example: Comparing Different Code Types

For symbols $\{A, B, C, D\}$ with probabilities $\{0.5, 0.25, 0.125, 0.125\}$:

Code Type	A	B	C	D	Property
Non-singular	0	1	00	11	Distinct but ambiguous: ”00” = AA or C?
Uniquely decodable	0	01	011	0111	Unique but need lookahead
Prefix code	0	10	110	111	Instant decoding: ”0” = A, stop
Optimal prefix	0	10	110	111	Also Huffman optimal

Decoding examples:

- **Prefix code ”010110”**: 0→A, 10→B, 110→C = ”ABC” (instant)
- **Uniquely decodable ”00111”**: Need to scan ahead to determine split
- **Non-singular ”00”**: Ambiguous! Could be ”AA” or ”C”

Key insight: Prefix codes sacrifice some flexibility in codeword lengths (must satisfy Kraft inequality) for the benefit of instantaneous decoding.

2.2 Basic Terminology and Notation

Definition

Alphabet

An *alphabet* $\mathcal{X}$ is a finite set of possible symbols. Examples:

- Binary alphabet: $\mathcal{X} = \{0, 1\}$
- English letters: $\mathcal{X} = \{\texttt{A}, \ldots, \texttt{Z}\}$
- Bytes: $\mathcal{X} = \{0, 1, \ldots, 255\}$

Definition

Symbol

A *symbol* is a single element drawn from an alphabet. For example, the letter `E` is a symbol from the English alphabet.

Definition

Random Variable

A *random variable* X is a function that assigns a symbol or value to each outcome in a sample space:

$$X : \Omega \to \mathcal{X}$$

- Ω: Sample space (e.g., all possible states of a data source)

- $\mathcal{X}$: Set of possible values (alphabet, e.g., $\{0,1\}$, ASCII characters)
- For each $\omega \in \Omega$, $X(\omega)$ is the value assigned to outcome ω

Example: Binary Source

- $\Omega = \{\text{emits } 0, \text{emits } 1\}$ (or could be more complex underlying physics)
- $\mathcal{X} = \{0,1\}$
- $X(\text{emits } 0) = 0$, $X(\text{emits } 1) = 1$
- Probabilities: $P(X = 0) = P(\{\omega : X(\omega) = 0\}) = p$, $P(X = 1) = 1 - p$

Why this matters for compression:

- The entropy $H(X)$ depends on the probability distribution induced by X
- For $x \in \mathcal{X}$: $P(X = x) = P(\{\omega \in \Omega : X(\omega) = x\})$
- $H(X) = -\sum_{x \in \mathcal{X}} P(X = x) \log_2 P(X = x)$

Definition

Source

A *source* is a process that generates a sequence of symbols $(X_1, X_2, X_3, \dots)$ according to some probability law. In this lecture, we assume discrete sources unless stated otherwise.

Definition

Message (or Sequence)

A *message* is a finite sequence of symbols generated by the source:

$$x^n = (x_1, x_2, \dots, x_n)$$

Compression algorithms operate on messages, not on individual symbols.

Definition

Code and Codewords

A *code* assigns a binary string (codeword) to each symbol or message.

- Source symbols → codewords (e.g., Huffman coding)
- Messages → bitstreams (e.g., arithmetic coding)

Definition

Block Length

The *block length* n is the number of source symbols grouped together and encoded as a unit. Larger block lengths generally allow better compression but increase delay and complexity.

Definition

Model

A *model* estimates the probabilities of symbols or sequences. Better models lead to better compression by reducing uncertainty.

2.3 Information and Redundancy: The Core Concepts

2.3.1 Information: A Formal Measure of Uncertainty Reduction

In information theory, information is defined rigorously as a **quantitative measure of the reduction in uncertainty** that results from observing the outcome of a random event.

Definition 2.1. *Let X be a random event that occurs with probability $p = \Pr(X)$. The* ***information content*** *(or* self-information*) $I(X)$ provided by the occurrence of X is defined as:*

$$I(X) = \log_b \left(\frac{1}{p}\right) = -\log_b(p)$$

where:

- $b = 2$ *yields* ***bits*** *(binary digits)*
- $b = e$ *yields* ***nats*** *(natural units)*
- $b = 10$ *yields* ***hartleys*** *or* ***dits***

Example

Predictability vs. Information:

- In a city where it rains every day, the statement “It rained today” conveys almost no information because it was expected
- A file that contains only the bit ‘1’ provides very little information
- A coin that always lands heads produces outcomes, but no information

  Key idea: Perfect predictability implies zero information gain.

Example

Daily Weather Forecast — Information Content:

- Sunny in Phoenix (probability 0.9): $I = -\log_2 0.9 \approx 0.15$ bits
- Snow in Phoenix (probability 0.001): $I = -\log_2 0.001 \approx 9.97$ bits
- Rain in Seattle (probability 0.3): $I = -\log_2 0.3 \approx 1.74$ bits

Interpretation: Rare events carry more information because they reduce uncertainty the most.

2.3.2 Redundancy: The Enemy of Information and the Friend of Compression

Redundancy refers to predictable or repeated structure in data. It is what allows data to be represented using fewer bits.

1. **Spatial Redundancy**: Neighboring data values are highly correlated

   Example

   In a photograph of a clear blue sky, most neighboring pixels have nearly identical color values.

   - **Naive**: Store the RGB value of each pixel independently
   - **Smarter**: Encode repeated pixel values using run-length encoding
   - **Even smarter**: Predict each pixel from its neighbors and encode only the small prediction error

2. **Statistical Redundancy**: Some symbols occur far more frequently than others

   Example

   English letter frequencies:

Letter	Frequency	Letter	Frequency
E	12.7%	Z	0.07%
T	9.1%	Q	0.10%
A	8.2%	J	0.15%

   Frequent letters get shorter codes in variable-length coding schemes.

3. **Knowledge Redundancy**: Information already known to both encoder and decoder

4. **Perceptual Redundancy**: Information that humans cannot perceive

2.4 Entropy: The Fundamental Limit

2.4.1 What is Entropy? Different Perspectives

Definition

Shannon Entropy of a discrete random variable X with possible values $\{x_1, x_2, \ldots, x_n\}$ having probabilities $\{p_1, p_2, \ldots, p_n\}$:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i \quad \text{bits}$$

Two Complementary Interpretations:

1. **Average Information Content**: Expected value of information content across all symbols
2. **Uncertainty or Surprise**: Measures how uncertain we are about the next symbol

2.4.2 Calculating Entropy: Step by Step

Example

Binary Source Example - Detailed Calculation:

Consider a biased coin: P(Heads) = 0.8, P(Tails) = 0.2

Step 1: **Calculate individual information content**:

$$I_H = -\log_2(0.8) \approx 0.3219 \text{ bits}$$
$$I_T = -\log_2(0.2) \approx 2.3219 \text{ bits}$$

Step 2: **Calculate entropy as expected value**:

$$H = 0.8 \times 0.3219 + 0.2 \times 2.3219 = 0.7219 \text{ bits}$$

Step 3: **Verify using direct formula**:

$$H = -[0.8 \log_2(0.8) + 0.2 \log_2(0.2)] \approx 0.7219 \text{ bits}$$

Key Insights:

- **Extreme cases**:

- Fair coin (P=0.5): $H = 1.0$ bit (maximum uncertainty)
- Always heads (P=1.0): $H = 0$ bits (no uncertainty)
- 90% heads: $H \approx 0.469$ bits

2.4.3 Entropy of English Text: A Practical Case Study

Example

Calculating English Letter Entropy:
Based on letter frequencies in typical English text:

$$H \approx 4.18 \text{ bits/letter}$$

Layered Interpretation:

- **First-order entropy (letters independent)**: 4.18 bits/letter
- **Actual uncertainty is lower**: Letters have dependencies (Q→U)
- **Comparison with encoding schemes**:

Encoding Method	**Bits/Letter**
Naive (5 bits for 26 letters)	5.00
Huffman (letter-based)	4.30
Using digram frequencies	3.90
Using word frequencies	2.30
Optimal with full context	∼1.50

2.4.4 Beyond First-Order Entropy: The Full Picture

Higher-Order Entropies quantify uncertainty while accounting for increasing context:

- **Zero-order entropy (H_0)**: $H_0 = \log_2 |\mathcal{X}|$
- **First-order entropy (H_1)**: $H_1 = -\sum p(x) \log_2 p(x)$
- **Second-order entropy (H_2)**: $H_2 = -\sum p(x, y) \log_2 p(x|y)$
- ***N*th-order entropy (H_N)**: $H_N = -\sum p(x_1, \ldots, x_N) \log_2 p(x_N \mid x_1, \ldots, x_{N-1})$

2.4.5 Entropy Rate

The **entropy rate** of a source is defined as the limiting uncertainty per symbol when arbitrarily long contexts are available:

$$H_\infty = \lim_{N \to \infty} H_N$$

2.4.6 The Entropy Theorem: Why It Matters

Definition

Expected Code Length

For a source with symbols $\{x_1, x_2, \ldots, x_n\}$ having probabilities $\{p_1, p_2, \ldots, p_n\}$, and a code that assigns codeword lengths $\{\ell_1, \ell_2, \ldots, \ell_n\}$, the **expected code length** L is:

$$L = \mathbb{E}[\ell(X)] = \sum_{i=1}^{n} p_i \ell_i \quad \text{(bits per symbol)}$$

This measures the average number of bits needed to encode one symbol from the source.

Definition

Compression Ratio and Efficiency

For a source with entropy $H(X)$ and code with expected length L:

- **Compression ratio**: $\rho = \frac{\text{original bits}}{\text{compressed bits}}$
- **Efficiency**: $\eta = \frac{H(X)}{L} \leq 1$
- **Redundancy**: $R = L - H(X) \geq 0$

Perfect compression occurs when $\eta = 1$ (100% efficient) and $R = 0$.

Important

Shannon's Source Coding Theorem (1948)

For a discrete memoryless source with entropy H and any $\epsilon > 0$:

1. **Converse (Impossibility Result)**:
   No lossless coding scheme can achieve expected code length $L < H$.

$$\boxed{L \geq H \quad \text{for any uniquely decodable code}}$$

2. **Achievability (Possibility Result)**:
   There exists a lossless coding scheme (specifically, block coding with suffi-

ciently large block size n) such that:

$$\boxed{H \leq L < H + \epsilon}$$

Equivalently: For any $\epsilon > 0$, $\exists n$ such that:

$$\frac{L_n}{n} < H + \epsilon$$

where L_n is the expected length for blocks of size n.

Interpretation:

- **Entropy is the fundamental limit**: H bits/symbol is the best we can ever do
- **We can get arbitrarily close**: With clever coding, we can approach this limit as closely as desired
- **The gap is achievable**: The "$+\epsilon$" represents practical overhead that can be made arbitrarily small

Example

Understanding the Theorem with Numbers
Consider a binary source with $p(0) = 0.9$, $p(1) = 0.1$:

$$H = -0.9 \log_2 0.9 - 0.1 \log_2 0.1 \approx 0.469 \text{ bits/symbol}$$

- **Naive coding**: Use 1 bit per symbol $\rightarrow L = 1.0$, efficiency $\eta = 0.469/1.0 = 46.9\%$
- **Huffman coding**: 0→0, 1→1 (same as naive!) $\rightarrow L = 1.0$, $\eta = 46.9\%$ *Why so bad?* Because we're coding symbols individually.
- **Block coding (n=2)**: Code pairs of symbols:

$$\begin{aligned}
00 &\rightarrow 0 \quad (\ell = 1, p = 0.81) \\
01 &\rightarrow 10 \quad (\ell = 2, p = 0.09) \\
10 &\rightarrow 110 \quad (\ell = 3, p = 0.09) \\
11 &\rightarrow 111 \quad (\ell = 3, p = 0.01)
\end{aligned}$$

$L_2 = 0.81 \times 1 + 0.09 \times 2 + 0.09 \times 3 + 0.01 \times 3 = 1.29$ bits/block Per symbol: $L = L_2/2 = 0.645$ bits/symbol, $\eta = 0.469/0.645 \approx 72.7\%$

- **Block coding (n=3)**: Would get even closer to 0.469
- **Theoretical limit**: As $n \to \infty$, $L \to 0.469$

Key insight: The theorem tells us:

1. We can never beat 0.469 bits/symbol (impossibility)
2. We can get as close as we want to 0.469 bits/symbol (achievability)

Important

What the Theorem Does NOT Say

- **It doesn't say how to construct the code** - just that one exists
- **It doesn't guarantee practical implementation** - block size n might need to be huge
- **It doesn't account for computational complexity** - the code might be too complex to implement
- **It assumes we know the true probabilities** - in practice, we estimate them

Yet, this theorem is revolutionary because it:

1. Establishes a **fundamental limit** (like the speed of light in physics)
2. Provides a **benchmark** for evaluating compression algorithms
3. Guides algorithm design toward this limit

2.4.7 Key Takeaways

- Entropy measures both **average information** and **uncertainty**
- Higher-order models reduce entropy by exploiting dependencies
- The entropy rate represents the ultimate compression limit
- Shannon's theorem precisely separates the *possible* from the *impossible*

2.5 Entropy as a Lower Bound

2.5.1 The Fundamental Inequality

Theorem 2.2 (Entropy Lower Bound). *For any **uniquely decodable** code C for source X:*

$$L(C) \geq H(X)$$

where $L(C) = \mathbb{E}[\ell(X)] = \sum_i p_i \ell_i$ is the expected code length.

Example

Binary Source with $p(0) = 0.9$, $p(1) = 0.1$

$$H = -0.9 \log_2 0.9 - 0.1 \log_2 0.1 \approx 0.469 \text{ bits/symbol}$$

Why we can't achieve 0.4 bits/symbol:

1. For 100 symbols, typical sequences: $2^{100 \times 0.469} \approx 2^{46.9}$
2. To encode all uniquely, need at least $2^{46.9}$ codewords
3. At 0.4 bits/symbol, total bits = 40
4. # codewords $\leq 2^{40} < 2^{46.9} \rightarrow$ impossible!

2.6 Kraft-McMillan Inequality: Core Theoretical Tool

2.6.1 Statement and Interpretation

Theorem 2.3 (Kraft-McMillan Inequality (Binary Case)). *Let $\ell_1, \ell_2, \ldots, \ell_m$ be the lengths of codewords in a **prefix code**. Then:*

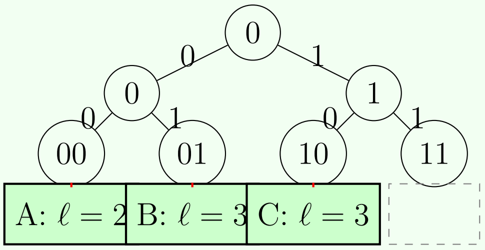
$$\sum_{i=1}^{m} 2^{-\ell_i} \leq 1$$

Conversely, if integers $\ell_1, \ldots, \ell_m$ satisfy this inequality, then there exists a binary prefix code with these lengths.

Example

Tree Visualization of Kraft Inequality

Consider a binary tree of depth $L = \max \ell_i$:



Calculating Kraft sum for $\{\ell_A = 2, \ell_B = 3, \ell_C = 3\}$:

$$\sum 2^{-\ell_i} = 2^{-2} + 2^{-3} + 2^{-3} = 0.25 + 0.125 + 0.125 = 0.5 \leq 1$$

Example

Testing Code Feasibility

1. **Valid lengths**: $\{1, 2, 3, 3\}$

$$\sum = 2^{-1} + 2^{-2} + 2^{-3} + 2^{-3} = 0.5 + 0.25 + 0.125 + 0.125 = 1.0 \quad \text{VALID}$$

2. **Invalid lengths**: $\{1, 1, 2\}$

$$\sum = 2^{-1} + 2^{-1} + 2^{-2} = 0.5 + 0.5 + 0.25 = 1.25 > 1 \quad \text{INVALID}$$

2.7 Optimality of Huffman Codes (Theory Only)

2.7.1 The Optimality Theorem

Theorem 2.4 (Huffman Optimality)**.** *Given a source with symbol probabilities* $p_1, p_2, \ldots, p_m$*, the Huffman algorithm produces a prefix code that* ***minimizes*** *the expected code length* $L = \sum_{i=1}^{m} p_i \ell_i$ *among all prefix codes.*

2.7.2 Relation to Entropy

For any Huffman code:

$$H(X) \leq L_{\text{Huffman}} < H(X) + 1$$

Example

Understanding the "+1" Gap

Consider source with probabilities $\{0.6, 0.3, 0.1\}$:

$$H \approx 1.295 \text{ bits}$$

Ideal (non-integer) lengths: $-\log_2 p_i = \{0.737, 1.737, 3.322\}$
Huffman code: 0.6→0, 0.3→10, 0.1→11

$$L = 0.6 \times 1 + 0.3 \times 2 + 0.1 \times 2 = 1.4 \text{ bits}$$

Comparison:

- Entropy: 1.295 bits
- Huffman: 1.400 bits
- Gap: 0.105 bits (much less than 1!)

2.7.3 Why Huffman is Optimal but Not Perfect

Important

Huffman is Optimal Within a Restricted Class
Huffman is optimal among:

- **Symbol-by-symbol** codes
- **Prefix** codes
- **Static** codes

But real optimality might require:

- **Block coding**
- **Fractional bits** (arithmetic coding)
- **Adaptive probabilities**

2.8 Limitations of Symbol-by-Symbol Coding

2.8.1 Three Fundamental Limitations

1. **Cannot Exploit Dependencies**
2. **Integer Length Constraint**: $\ell_i \in \mathbb{Z}^+$ but $-\log_2 p_i \in \mathbb{R}$
3. **Memoryless Assumption**

Example

English Text: The Cost of Symbol-by-Symbol

- **First-order entropy** (ignoring dependencies): 4.0 bits/letter
- **Actual entropy rate** (with dependencies): 1.5 bits/letter
- **Huffman on letters**: 4.0 bits/letter
- **Gap**: 2.5 bits/letter wasted due to ignoring dependencies

2.9 Block Coding and Improved Efficiency

2.9.1 The Block Coding Idea

Instead of coding symbols individually, group them into blocks of length n:

$$\mathbf{X} = (X_1, X_2, \ldots, X_n)$$

Definition

nth Extension of a Source
For a source with alphabet $\mathcal{X}$, the nth extension has alphabet:

$$\mathcal{X}^n = \{(x_1, \ldots, x_n) : x_i \in \mathcal{X}\}$$

with size $|\mathcal{X}|^n$.

2.9.2 Key Mathematical Results

Theorem 2.5 (Entropy of Block Source)**.** *For a discrete memoryless source:*

$$H(X^n) = nH(X)$$

Theorem 2.6 (Block Coding Performance)**.** *There exists a prefix code C_n for X^n such that:*

$$nH(X) \leq L_n < nH(X) + 1$$

Dividing by n:

$$H(X) \leq \frac{L_n}{n} < H(X) + \frac{1}{n}$$

Important

The Magic of Block Coding

As $n \to \infty$:

$$\frac{L_n}{n} \to H(X)$$

We can approach entropy **arbitrarily closely** by making blocks larger!

2.9.3 Step-by-Step Example

Example

Binary Source: $p(0) = 0.9$, $p(1) = 0.1$, $H \approx 0.469$

Step 1: $n = 1$ **(symbol-by-symbol)**

- Huffman: 0→0, 1→1
- $L_1 = 1$ bit/symbol
- Efficiency: $\eta = 0.469/1 = 46.9\%$

Step 2: $n = 2$ **(code pairs)**

- Block probabilities: $P(00) = 0.81$, $P(01) = 0.09$, $P(10) = 0.09$, $P(11) = 0.01$
- Codes: 00→0, 01→10, 10→110, 11→111
- Per symbol: $L_2/2 = 0.645$ bits/symbol
- Efficiency: $\eta = 0.469/0.645 = 72.7\%$

2.10 Trade-Offs in Block Coding

2.10.1 The Engineering Challenges

1. **Exponential Alphabet Growth**: $|\mathcal{X}^n| = |\mathcal{X}|^n$
2. **Memory Requirements**: Huffman tree has $2m^n - 1$ nodes
3. **Computational Complexity**: $O(m^n \log m^n)$
4. **Delay and Latency**: Must wait for n symbols

2.11 From Block Coding to Modern Compression

2.11.1 Arithmetic Coding: Fractional-Bit Block Coding

Important

Arithmetic Coding as "Infinite Block Coding"
Arithmetic coding cleverly avoids the exponential growth problem:

- **Idea**: Encode entire message as a single real number in [0,1)
- **No explicit blocks**: Processes symbols sequentially
- **Fractional bits**: Achieves $L \approx H(X)$ without large n
- **Removes integer constraint**: No "+1" overhead!

2.11.2 Context Modeling: Approximating Large Blocks

Instead of explicit block coding, modern compressors use:

1. **Context Models**: Predict next symbol based on previous k symbols
2. **Prediction + Residual Coding**: Encode only prediction error
3. **Dictionary Methods (LZ family)**: Build dictionary of previously seen phrases

2.12 What Theory Guarantees vs What Practice Achieves

Aspect	Theory Guarantees	Practice Achieves
Optimality	Can approach entropy arbitrarily closely	Gets close, but with practical limits
Block Size	$n \to \infty$ gives optimality	n limited by memory, latency, complexity
Complexity	Ignored, infinite resources allowed	Critical constraint; often dominates design

2.13 Summary and Key Takeaways

Important

Five Fundamental Lessons

1. **Entropy is the Absolute Limit**: $L \geq H(X)$ for any lossless code

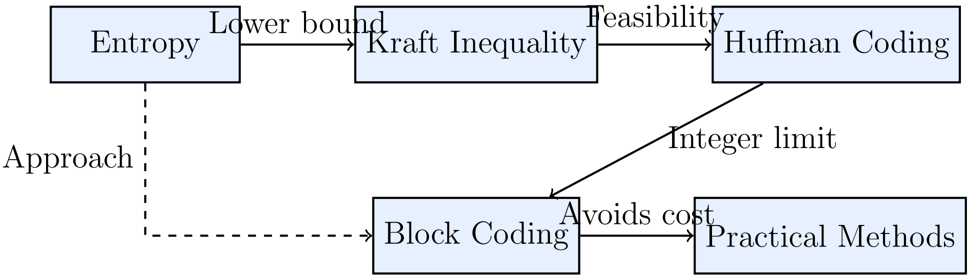
2. **Kraft-McMillan Constrains All Codes**: $\sum 2^{-\ell_i} \leq 1$

3. **Huffman is Optimal Among Prefix Codes**: But limited by integer lengths

4. **Block Coding Allows Approaching Entropy**: $\lim_{n\to\infty} \frac{L_n}{n} = H(X)$

5. **Practical Compression Balances Efficiency and Complexity**

2.13.1 The Big Picture



2.13.2 Looking Forward

- **Next lecture: Arithmetic Coding**: Removes integer constraint
- **Then: Dictionary Methods (LZ family)**: Adaptive to data statistics
- **Finally: Modern Compressors**: Combining multiple techniques

Final Thought

Shannon's 1948 paper told us *exactly how good compression could possibly be.* Every compressor since has been trying to approach that limit while staying within practical constraints.

The gap between theory and practice is where engineering creativity lives!

3: Lecture 3: Advanced Entropy Coding & Extensions

Lecture 3: Beyond Huffman – Advanced Entropy Coding Methods

3.1 Introduction & Motivation

Important

Recall Huffman Coding Limitations:

- **Integer code lengths**: Cannot reach entropy bound for highly skewed distributions
- **Static vs. Adaptive**: Standard Huffman requires prior knowledge of probabilities
- **Codebook overhead**: Need to transmit/store the coding tree
- **Symbol-by-symbol constraint**: Processes one symbol at a time

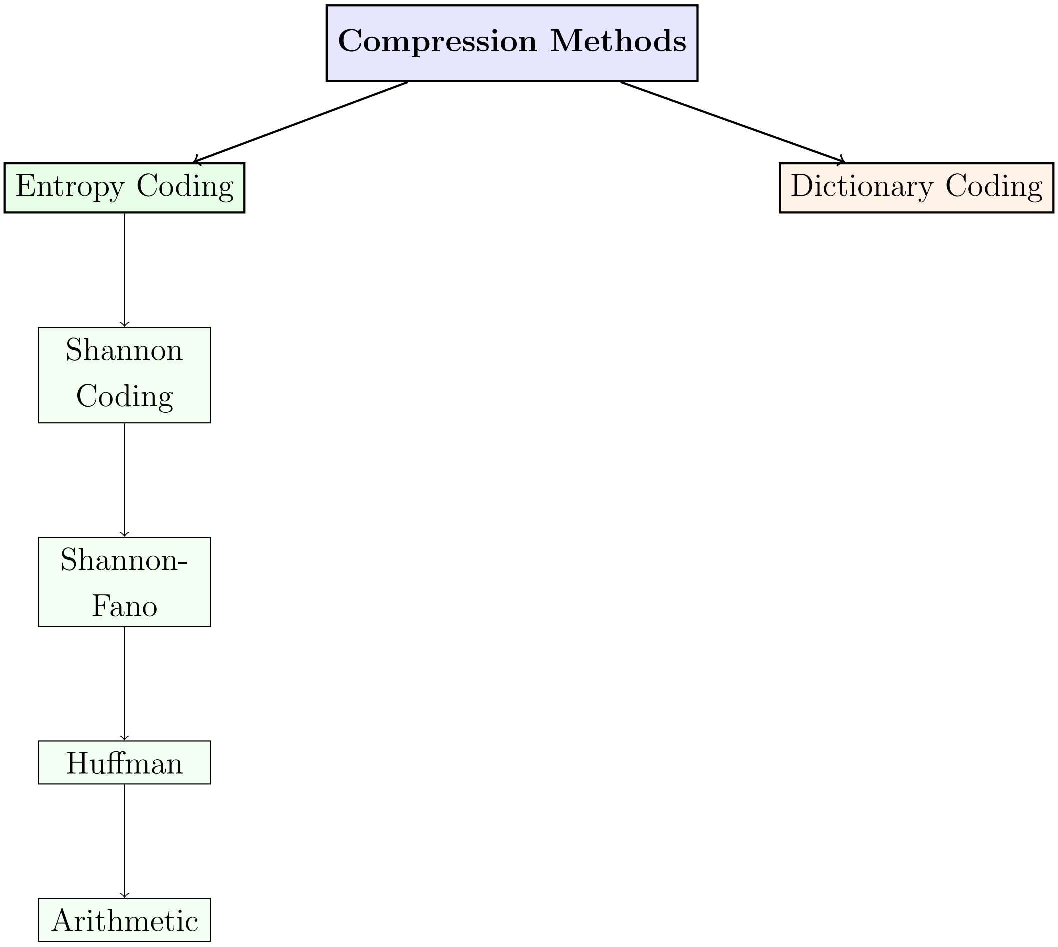
Lecture Roadmap:

1. **Framework**: Coding taxonomy and conceptual organization
2. **Historical methods**: Shannon & Shannon-Fano coding
3. **Practical improvements**: Canonical and Adaptive Huffman
4. **Next generation**: Arithmetic coding paradigm
5. **Synthesis**: Comparison and forward look

3.2 Coding Taxonomy & Framework

Definition

Coding Taxonomy: Classification of compression methods based on key characteristics



Key Dimensions in Compression Algorithm Design

1. **Modeling vs. Coding (Two-Stage View)**:
   - **Modeling Phase**: Analyzes data to estimate symbol probabilities or discover patterns.
     - *Examples*: Frequency counting (Huffman), context modeling (PPM), dictionary construction (LZ77)
   - **Coding Phase**: Converts modeled information into actual bits.
     - *Examples*: Huffman codes, arithmetic codes, LZ77 pointers
   - Some algorithms intertwine both (e.g., LZW builds dictionary while coding).
2. **Knowledge of Source Distribution**:
   - **Static/Fixed**: Uses a predefined model that doesn't change.
     - *Example*: JPEG Huffman tables, known language frequencies
     - Requires prior knowledge of data; fails if distribution differs.
   - **Adaptive**: Learns and updates the model during compression.
     - *Example*: Adaptive Huffman, LZ78 dictionary building

  - No prior knowledge needed; overhead for model transmission.
- **Universal**: Can compress any source asymptotically optimally.
  - *Theoretical property*: LZ family, arithmetic with adaptive model
  - Note: Most adaptive methods are universal in practice.

3. **Processing Granularity**:

   - **Symbol-by-Symbol**: Each input symbol maps to one codeword.
     - *Example*: Huffman coding
     - Simple but limited to integer bits per symbol.
   - **Block Coding**: Fixed-size groups of symbols coded together.
     - *Example*: Block-sorting (BWT) processes blocks
     - Can capture inter-symbol dependencies within block.
   - **Stream/Incremental**: Continuous processing with immediate output.
     - *Example*: Arithmetic coding, LZ77 sliding window
     - No blocking delay; good for real-time applications.
   - *Note*: "Message-wide" (whole file as one symbol) is theoretical ideal; arithmetic coding approximates it by treating the entire stream as one long fractional code.

4. **Algorithmic Approach (Primary Taxonomy)**:

   - **Statistical Coding**: Uses probability estimates (Huffman, Arithmetic)
   - **Dictionary Coding**: Replaces repeated patterns with references (LZ family)
   - **Transform Coding**: Changes data domain then codes (DCT, wavelet)
   - **Predictive Coding**: Predicts next value, codes difference (DPCM, LPC)

Important Relationships:

- Adaptive methods are usually universal for practical purposes.
- Stream coding is possible with both symbol-by-symbol (Huffman) and message-wide approaches (arithmetic).
- Block processing (like BWT) is often followed by stream coding (like MTF+RLE+arithmetic in bzip2).
- Modeling and coding can be separated (PPM + arithmetic) or combined (LZW).

3.3 Shannon Coding (1948)

Definition

Shannon Coding: A constructive method derived from Shannon's source coding theorem that assigns codewords by taking the binary expansion of cumulative probabilities:

$$l_i = \lceil -\log_2 p_i \rceil \quad \text{and} \quad \text{code}_i = \text{First } l_i \text{ bits of } F_i$$

where p_i is the probability of symbol i, and $F_i = \sum_{j=1}^{i-1} p_j$ is the cumulative probability of symbols ordered by decreasing probability.

Shannon Coding Algorithm

Input: Symbols $S = \{s_1, s_2, \dots, s_n\}$ with corresponding probabilities $P = \{p_1, p_2, \dots, p_n\}$
Output: Prefix-free binary code for each symbol

1. **Sort** symbols in non-increasing order of probability: $p_1 \geq p_2 \geq \cdots \geq p_n$
2. **Calculate codeword lengths:** For each symbol i, compute $l_i = \lceil -\log_2 p_i \rceil$
3. **Compute cumulative probabilities:**

$$F_1 = 0, \quad F_i = \sum_{j=1}^{i-1} p_j \quad \text{for } i = 2, \dots, n$$

4. **Generate codewords:** For each symbol i:

   a) Convert F_i to binary fractional representation (0.xxxx...)

   b) Take the first l_i bits after the binary point as the codeword

Example

Example: Given symbols with probabilities:

Symbol	Probability
A	0.5
B	0.25
C	0.125
D	0.125

Step-by-step construction:

1. **Sort by probability:** Already in non-increasing order ($0.5 \geq 0.25 \geq 0.125 \geq 0.125$)
2. **Calculate lengths:**

- $l_A = \lceil -\log_2 0.5 \rceil = \lceil 1.0 \rceil = 1$
- $l_B = \lceil -\log_2 0.25 \rceil = \lceil 2.0 \rceil = 2$
- $l_C = \lceil -\log_2 0.125 \rceil = \lceil 3.0 \rceil = 3$
- $l_D = \lceil -\log_2 0.125 \rceil = \lceil 3.0 \rceil = 3$

3. **Compute cumulative probabilities:**

   - $F_A = 0$ (first symbol)
   - $F_B = 0.5$ (just A's probability)
   - $F_C = 0.5 + 0.25 = 0.75$ (A + B)
   - $F_D = 0.5 + 0.25 + 0.125 = 0.875$ (A + B + C)

4. **Convert F_i to binary and take first l_i bits:**

   - **A:** $F_A = 0.0_{10}$ in binary is $0.0000\ldots_2$
     - Take first $l_A = 1$ bit: **0**
   - **B:** $F_B = 0.5_{10}$ in binary is $0.1000\ldots_2$
     - Take first $l_B = 2$ bits: **10**
   - **C:** $F_C = 0.75_{10}$ in binary is $0.1100\ldots_2$
     - Take first $l_C = 3$ bits: **110**
   - **D:** $F_D = 0.875_{10}$ in binary is $0.1110\ldots_2$
     - Take first $l_D = 3$ bits: **111**

Resulting code:

Symbol	Probability	l_i	F_i	Shannon Code
A	0.5	1	0.0	0
B	0.25	2	0.5	10
C	0.125	3	0.75	110
D	0.125	3	0.875	111

Expected length: $L = 0.5 \times 1 + 0.25 \times 2 + 0.125 \times 3 + 0.125 \times 3 = 1.75$ bits/symbol

Important

Critical Note on Sorting:
The sorting step is **essential** in Shannon coding because:

- Cumulative probabilities F_i depend on the order of symbols

- Sorting ensures intervals in [0,1) are assigned in decreasing size order
- Without sorting, the resulting code may not be prefix-free
- Sorting guarantees the length condition $\frac{1}{2^{l_i}} \leq p_i < \frac{1}{2^{l_i-1}}$ leads to non-overlapping intervals

Why it works: When probabilities are sorted, each symbol's interval of size p_i starts at F_i and ends at $F_i + p_i$. The codeword length ensures the binary expansion of F_i to l_i bits uniquely identifies the starting point without overlapping with neighboring intervals.

Understanding the Binary Expansion Process

When we write F_i in binary (e.g., $0.5 = 0.1_2$, $0.75 = 0.11_2$), we're essentially:

- Dividing the interval [0,1) into subintervals based on probabilities
- Each symbol gets an interval of size p_i
- The codeword is the **binary fraction** representing the **start** of that interval
- We use exactly $l_i = \lceil -\log_2 p_i \rceil$ bits, which ensures:

$$\frac{1}{2^{l_i}} \leq p_i < \frac{1}{2^{l_i-1}}$$

- This guarantees unique prefixes because intervals don't overlap!

Important

Properties of Shannon Coding:

- **Constructive proof**: Demonstrates that prefix codes exist for any lengths satisfying Kraft inequality
- **Simple to compute**: Direct from probabilities, no tree needed
- **Requires sorting**: Symbols must be ordered by decreasing probability first
- **Not optimal**: Unlike Huffman, doesn't minimize expected length (compare: Huffman would give A=0, B=10, C=110, D=111 **same in this case!**)
- **Theoretical importance**: Foundation for Shannon's source coding theorem
- **Efficiency bound**: $H(X) \leq L < H(X) + 1$ (like Shannon's theorem says)

Example

Counterexample: What happens if we don't sort?
Consider the same symbols but in different order: C (0.125), A (0.5), D (0.125), B (0.25)
Without sorting:

- $F_C = 0$, $l_C = 3$ → codeword: 000
- $F_A = 0.125$, $l_A = 1$ → codeword: 0 (binary: 0.001... → first bit is 0)
- $F_D = 0.625$, $l_D = 3$ → codeword: 101
- $F_B = 0.75$, $l_B = 2$ → codeword: 11

Problem: Codeword "0" (for A) is a prefix of "000" (for C) → **not prefix-free!** This demonstrates why sorting is essential in Shannon coding.

3.4 Shannon–Fano Coding (1949)

Definition

Shannon–Fano Coding: A top-down, recursive source coding technique that assigns binary codewords by repeatedly partitioning a set of symbols into two subsets whose total probabilities are as close as possible. The method was developed independently by *Claude Shannon* and *Robert Fano* in 1949.

Algorithm Description

High-level idea: Symbols with higher probabilities should receive shorter codewords. This is achieved by repeatedly splitting the symbol set into two probability-balanced groups and assigning binary prefixes.

Step-by-step procedure:

1. **Sort** the symbols in decreasing order of probability:
$$p_1 \geq p_2 \geq \cdots \geq p_n.$$
2. **Recursive partitioning:**
   - If the current set contains only one symbol, stop (this is the base case).

- Find an index k that minimizes

$$\left|\sum_{i=1}^{k} p_i - \sum_{i=k+1}^{n} p_i\right|.$$

- This divides the symbols into two subsets:

$$S_1 = \{1, \dots, k\}, \qquad S_2 = \{k+1, \dots, n\}.$$

- Append bit **0** to the codewords of all symbols in S_1.
- Append bit **1** to the codewords of all symbols in S_2.
- Apply the same procedure recursively to S_1 and S_2.

Example with Six Symbols

Example

Example: Consider six symbols with the following probabilities.

Symbol	Probability	$-\log_2 p_i$
A	0.30	1.74
B	0.25	2.00
C	0.20	2.32
D	0.10	3.32
E	0.10	3.32
F	0.05	4.32

Construction process

Step 1: First split (balance 0.55 vs. 0.45)

- Sorted symbols: A(0.30), B(0.25), C(0.20), D(0.10), E(0.10), F(0.05)
- Best split: $\{A, B\}$(0.55) and $\{C, D, E, F\}$(0.45)
- Prefix assignment: $\{A, B\} \to 0$, $\{C, D, E, F\} \to 1$

Step 2: Split $\{A, B\}$

- A: **00**, B: **01**

Step 3: Split $\{C, D, E, F\}$

- Best split: $\{C\}$(0.20) and $\{D, E, F\}$(0.25)
- C: **10**, $\{D, E, F\} \to 11$

Step 4: Split $\{D, E, F\}$

- Best split: $\{D\}(0.10)$ and $\{E, F\}(0.15)$
- D: **110**, $\quad \{E, F\} \to 111$

Step 5: Split $\{E, F\}$

- E: **1110**, $\quad$ F: **1111**

Final codes:

Symbol	Probability	Code	Length
A	0.30	00	2
B	0.25	01	2
C	0.20	10	2
D	0.10	110	3
E	0.10	1110	4
F	0.05	1111	4

Expected code length:

$$L = 2.40 \text{ bits/symbol.}$$

Entropy:

$$H(X) \approx 2.25 \text{ bits/symbol.}$$

Efficiency:

$$\frac{H(X)}{L} \approx 93.8\%.$$

Visual Tree Representation

```
                 ROOT
                /    \
              0/      \1
              /        \
       {A,B} 0.55    {C,D,E,F} 0.45
          / \            / \
        0/   \1        0/   \1
        /     \        /     \
{A} 0.30  {B} 0.25  {C} 0.20  {D,E,F} 0.25
  00        01        10         / \
                               0/   \1
                               /     \
                         {D} 0.10  {E,F} 0.15
                           110        / \
                                    0/   \1
                                    /     \
                              {E} 0.10  {F} 0.05
                                1110      1111
```

Key Observations

- **Probability-balanced splits:** The algorithm focuses on balancing probabilities rather than the number of symbols.
- **Variable code lengths:** More probable symbols receive shorter codewords.
- **Prefix-free property:** No codeword is a prefix of another.
- **Near-optimal performance:** The efficiency is high but not guaranteed to be optimal.

Important

Limitations and Historical Context

- Shannon–Fano coding does *not* always produce the optimal code.
- Huffman coding (1952) guarantees the minimum average code length.
- Historically important as a precursor to Huffman coding.

3.5 Canonical Huffman Codes

Definition

Canonical Huffman Code: A standardized representation of a Huffman code in which:

- Codes are assigned in lexicographic (binary) order
- All codewords of the same length are consecutive binary numbers
- The first codeword of each length is the smallest possible binary value
- Only the code lengths are required to reconstruct the entire code

This enables compact storage and fast table-based decoding.

Why Canonical Huffman Codes?

A standard Huffman algorithm produces *optimal code lengths*, but the exact bit patterns depend on implementation details such as tie-breaking and tree construction:

- Different trees can yield the same optimal set of code lengths
- Different bit assignments, but identical compression performance

Canonical Huffman coding fixes **one unique and deterministic assignment** for a given set of code lengths so that:

1. Only code lengths need to be transmitted
2. The decoder can reconstruct the codes without ambiguity
3. Decoding can be implemented efficiently using lookup tables

Two-Step Process

Complete workflow:

1. **Run the standard Huffman algorithm** to obtain optimal code lengths l_i
2. **Apply the canonical transformation** to convert lengths into standardized codes

Symbol probabilities $\xrightarrow{\text{Huffman}}$ Code lengths $\xrightarrow{\text{Canonical}}$ Canonical codes

Canonical Huffman Construction Algorithm

Input: Code lengths l_i obtained from a standard Huffman algorithm **Output:** Canonical Huffman codes

1. **Sort symbols** in ascending order of code length l_i (primary key), and by symbol order or value (secondary key for tie-breaking)
2. **Count symbols per length:**
   - Let $l_{\min}$ and $l_{\max}$ be the minimum and maximum code lengths
   - For each length l, compute $\text{count}[l]$ = number of symbols with length l
3. **Compute starting codes:**
   - $\text{start\_code}[l_{\min}] = 0$
   - For $l = l_{\min} + 1$ to $l_{\max}$:

$$\text{start\_code}[l] = (\text{start\_code}[l-1] + \text{count}[l-1]) \times 2$$

4. **Initialize assignment pointers:**

$$\text{next\_code}[l] = \text{start\_code}[l] \quad \text{for all } l$$

5. **Assign codes to symbols** (in sorted order):
   - For each symbol i with length l_i:

  - Assign next_code[l_i] as its code (converted to l_i-bit binary)
  - Increment next_code[l_i] by 1

Understanding the Shift Operation: Why Multiply by 2?

The key step start_code[l] = (start_code[$l-1$] + count[$l-1$]) × 2 ensures:

Important

Visual Intuition: Imagine all codes of length $l-1$ are written as binary numbers. When we multiply by 2 (left shift by 1 bit), we:

- Append a 0 to each existing code
- Create room for new codes that are one bit longer
- Ensure the new codes maintain prefix-free property

Example: If the last code of length 2 is "11" (binary 3), then:

$$(\text{last code} + 1) \times 2 = (3+1) \times 2 = 8 = 1000_{\text{binary}}$$

The first code of length 3 is "100" (3 bits of 1000), which follows "11" properly.

Mathematical Explanation:

- start_code[$l-1$] + count[$l-1$] gives the **first unused code value** after all codes of length $l-1$
- Multiplying by 2 (left shift by 1 bit) makes room for an extra bit
- This ensures all codes of length l start with a 0 in their new bit position
- The result is the smallest valid starting point for the next length group

Bit Pattern Visualization:

```
Length 2 codes: 00, 01, 10, 11
Length 3 codes start at: (11 + 1) × 2 = 1000_binary
So length 3 codes are: 100, 101, 110, 111
Notice: 11 is NOT a prefix of 100!
```

Complete Worked Example

Example

Example: Suppose the Huffman algorithm produces the following code lengths:

Symbol	Length (l_i)
A	2
B	3
C	3
D	3
E	4
F	4

Key Point: The code lengths (l_i) are already determined by the Huffman tree construction. These lengths tell us *exactly* how many bits each symbol's code will use.

Step 1: Sort symbols by length (primary), then symbol order (secondary)

- **Critical:** Sorting ensures consistent code assignment
- Order: A (2), B (3), C (3), D (3), E (4), F (4)
- Within same length group, maintain original symbol order

Step 2: Count symbols per length

$$\text{count}[2] = 1, \quad \text{count}[3] = 3, \quad \text{count}[4] = 2$$

Step 3: Compute starting codes (with shift explanation)

$$\begin{aligned}
\text{start\_code}[2] &= 0 \quad \text{(represented as 00 in binary, using \textbf{2 bits} because length=2)} \\
\text{start\_code}[3] &= (\text{start\_code}[2] + \text{count}[2]) \times 2 \\
&= (0 + 1) \times 2 = 2 \quad \text{(represented as 010 in binary, using \textbf{3 bits} because length=3)} \\
&\quad \textit{Interpretation: After 1 code of length 2, shift left (add 0 bit) for length 3} \\
\text{start\_code}[4] &= (\text{start\_code}[3] + \text{count}[3]) \times 2 \\
&= (2 + 3) \times 2 = 10 \quad \text{(represented as 1010 in binary, using \textbf{4 bits} because length=4)} \\
&\quad \textit{Interpretation: After 3 codes of length 3, shift left (add 0 bit) for length 4}
\end{aligned}$$

Important: The multiplication by 2 is a **left shift** operation. It adds one more bit to the code while maintaining the prefix-free property.

Step 4: Initialize assignment pointers

$$\text{next\_code}[2] = 0, \quad \text{next\_code}[3] = 2, \quad \text{next\_code}[4] = 10$$

Step 5: Assign codes (in sorted order)

Symbol	Length	Decimal	Binary Code	next_code after
A	2	0	`00`	1
B	3	2	`010`	3
C	3	3	`011`	4
D	3	4	`100`	5
E	4	10	`1010`	11
F	4	11	`1011`	12

How binary conversion works:

- For symbol A (length=2, decimal=0): Binary: 00 (padded to 2 bits)
- For symbol B (length=3, decimal=2): Binary: 010 (2 in binary is 10, padded to 3 bits: 010)
- For symbol E (length=4, decimal=10): Binary: 1010 (10 in binary is 1010, already 4 bits)

Algorithm Summary:

1. **Input**: Code lengths l_i for each symbol (from Huffman tree)
2. **For each length** l:
   - First code of length l: start_code[l]
   - Subsequent codes: increment by 1
   - All codes use exactly l bits
3. **Key formula**: start_code[$l+1$] = (start_code[l] + count[l]) × 2

Final canonical codes (verify properties):

Symbol	Length	Canonical Code	Check
A	2	00	✓ First length 2 code
B	3	010	✓ First length 3 code
C	3	011	✓ Consecutive with B
D	3	100	✓ Consecutive with C
E	4	1010	✓ First length 4 code
F	4	1011	✓ Consecutive with E

Verification:

- All codes of same length are consecutive binary numbers ✓
- Codes are lexicographically ordered ✓
- No code is a prefix of another ✓

What If We Don't Sort? A Counterexample

Example

Problem without sorting: Suppose we have the same code lengths but assign codes without proper sorting:

Symbol	Length
D	3
A	2
E	4
B	3
F	4
C	3

If we assign codes in this random order using the same algorithm:

- D (length 3) gets code 010
- A (length 2) gets code 00
- E (length 4) gets code 1000
- **Problem:** Code "00" (A) is a prefix of "1000" (E)!

Conclusion: Sorting ensures that shorter codes are assigned first, preventing prefix conflicts between codes of different lengths.

Transmission and Decoding

Transmission format (very compact):

- Transmit only the sequence of code lengths in symbol order:

$$\langle l_1, l_2, \ldots, l_n \rangle$$

- Example for our symbols: $\langle 2, 3, 3, 3, 4, 4 \rangle$
- Each length can be represented using $\lceil \log_2 l_{\max} \rceil$ bits

Decoder reconstruction:

1. Read code lengths for all symbols (in original symbol order)
2. Sort symbols by (l_i, index) to match encoder's order
3. Reconstruct canonical codes using the same algorithm
4. Build decoding tables for fast lookup

Comparison with Standard Huffman

Standard Huffman (tree-based)	**Canonical Huffman (length-based)**
Output: Huffman tree structure	Output: Code lengths only
Must transmit tree (complex)	Transmit only lengths (simple)
Decoding by tree traversal (slower)	Decoding by table lookup (faster)
Multiple equivalent trees possible	Single deterministic assignment
Same optimal compression	Same optimal compression
More complex implementation	Simple, robust implementation
No sorting required	**Requires sorting by length**

Key Properties and Applications

- **Optimality:** Identical compression ratio to standard Huffman
- **Compactness:** Only code lengths stored/transmitted
- **Speed:** Table-based decoding is significantly faster
- **Standardization:** Used in DEFLATE (gzip/ZIP), JPEG, PNG, MPEG
- **Deterministic:** Same lengths always produce same codes

Important

Critical Insights:

1. Canonical Huffman is **not** a different compression algorithm from Huffman
2. The Huffman algorithm determines *how long* each code should be
3. The canonical transformation determines the *exact bit patterns* consistently
4. **Sorting by length** is essential to ensure prefix-free property
5. The ×2 operation (left shift) ensures proper separation between different length groups

3.6 Adaptive Huffman Coding

Overview

Definition

Adaptive Huffman Coding is a single-pass, lossless data compression technique in which the Huffman model is updated dynamically as symbols are encoded and decoded. The Huffman tree evolves on-the-fly, allowing both the encoder and decoder to adapt to changing symbol statistics without any prior knowledge of the source distribution.

Unlike static Huffman coding, adaptive Huffman coding does not require a preprocessing phase to compute symbol frequencies. Instead, symbol statistics are learned incrementally as the data stream is processed.

Limitations of Static Huffman Coding

Static Huffman coding assumes that symbol statistics are known in advance:

- Requires two passes over the data:
    1. Collect symbol frequencies
    2. Encode using the constructed Huffman tree
- The Huffman tree (or code lengths) must be transmitted as side information
- Cannot adapt to non-stationary or evolving symbol distributions

These limitations motivate adaptive schemes when symbol statistics are unknown or change over time.

Key Principle of Adaptive Huffman Coding

Adaptive Huffman coding maintains a Huffman tree that is updated after encoding or decoding each symbol.

Key invariant: After processing each symbol, the encoder and decoder maintain *identical Huffman trees* by applying the same updates in the same order. This ensures correct decoding without transmitting the tree explicitly.

Initialization

The algorithm begins with a special symbol:

- A single **NYT** (Not Yet Transmitted) node

When a symbol appears for the first time:

- The code for the NYT node is output
- The raw symbol value is transmitted
- The NYT node is expanded into:
  - A new NYT node
  - A leaf node representing the new symbol

Note: Some variants preinitialize the tree if the alphabet is fixed, but the standard adaptive Huffman algorithm begins with only the NYT node.

Tree Update Mechanism

After each symbol is processed:

- The frequency (weight) of the corresponding leaf node is incremented
- The tree is updated to preserve the **sibling property**

Definition

Sibling Property: Nodes are numbered such that nodes with higher weights have higher numbers. For any given weight, all nodes with that weight form a contiguous block. Each internal node has exactly two children.

To restore this property, nodes may be swapped with others of equal weight, followed by incrementing parent node weights. This update process proceeds bottom-up toward the root.

Encoding and Decoding Process

- If the symbol has appeared before:
  - Output its current Huffman code
- If the symbol is new:
  - Output the code for the NYT node
  - Output the symbol in raw (fixed-length) form
- Update the tree identically at both encoder and decoder

Note: Exact bit patterns depend on the update order and implementation. Adaptive Huffman coding guarantees prefix-free optimality but not unique codes.

Algorithms

Two classical adaptive Huffman algorithms are widely used:

- **FGK Algorithm** (Faller–Gallager–Knuth)
- **Vitter's Algorithm (Algorithm V)**, which improves worst-case performance and is commonly preferred

Both algorithms maintain the sibling property while ensuring efficient updates.

Performance Characteristics

- Update cost is **amortized** $O(1)$ per symbol
- Worst-case update time is proportional to the tree height
- Compression efficiency:
  - Initially suboptimal
  - Converges toward entropy-optimal coding as statistics stabilize

Comparison with Static Huffman Coding

Feature	**Static Huffman**	**Adaptive Huffman**
Number of passes	Two	One
Prior statistics required	Yes	No
Model transmission	Required	Not required
Adaptation to changes	No	Yes
Initial efficiency	High	Low
Asymptotic efficiency	Optimal	Near-optimal

Applications

Adaptive Huffman coding is well suited for:

- Streaming data sources
- Real-time communication
- Situations where full statistics cannot be collected in advance

However, it may be less suitable when symbol distributions are known and stable, or when computational simplicity is critical.

Summary

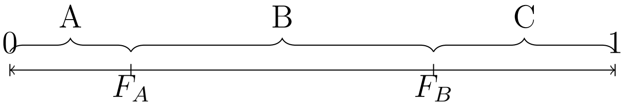
Adaptive Huffman coding extends classical Huffman coding to dynamic environments. By updating the coding model incrementally and maintaining strict structural invariants, it enables efficient, single-pass compression without prior knowledge of source statistics.

3.7 Arithmetic Coding: The Paradigm Shift

Definition

Arithmetic Coding: Encodes an entire message into a single fractional number in the interval $[0, 1)$, approaching the entropy bound very closely.

Core Idea: Represent messages as subintervals of $[0, 1)$:



Algorithm 1 Arithmetic Encoding Algorithm

```
Require: Message $m = s_1 s_2 \ldots s_k$, symbol probabilities $p_i$
Ensure: Final interval $[low, high)$
$low \leftarrow 0.0$, $high \leftarrow 1.0$
for each symbol $s$ in $m$ do
    $range \leftarrow high - low$
    $high \leftarrow low + range \times F_s$            ▷ $F_s$: cumulative prob up to $s$
    $low \leftarrow low + range \times F_{s-1}$         ▷ $F_{s-1}$: cumulative prob before $s$
end forreturn any number in $[low, high)$
```

Example

Example: Encode message "CAB" with probabilities: A(0.5), B(0.25), C(0.25)

Symbol	Probability	Cumulative
A	0.5	0.5
B	0.25	0.75
C	0.25	1.0

Encoding:

1. Start: $[0, 1)$
2. Process 'C': $[0.75, 1.0)$ (C occupies $[0.75, 1.0)$)
3. Process 'A': $range = 0.25$
    - $low = 0.75 + 0.25 \times 0.0 = 0.75$

- $high = 0.75 + 0.25 \times 0.5 = 0.875$
- New interval: $[0.75, 0.875)$

4. Process 'B': $range = 0.125$

- $low = 0.75 + 0.125 \times 0.5 = 0.8125$
- $high = 0.75 + 0.125 \times 0.75 = 0.84375$
- Final interval: $[0.8125, 0.84375)$

Output: Any number in $[0.8125, 0.84375)$, e.g., 0.8125 in binary

Important

Practical Implementation Issues:

- **Finite precision**: Use integer arithmetic with scaling
- **Renormalization**: Output bits when interval confined to one half
- **Carry-over**: Handle when interval spans midpoint
- **Termination**: Need special end-of-message symbol

Adaptive Arithmetic Coding: Easier than adaptive Huffman - just update probabilities as you go!

3.8 Comparison & Synthesis

Method	**Optimal?**	**Adaptive?**	**Complexity**	**Near Entropy?**	**Key Applications**
Shannon Coding	No	No	Low	No	Theoretical proofs
Shannon-Fano	No	No	Low	Sometimes	Historical
Huffman	Yes*	No	Low	Moderate	General purpose
Canonical Huffman	Yes*	No	Low	Moderate	DEFLATE, JPEG, PNG
Adaptive Huffman	Yes*	Yes	Medium	Moderate	Early compressors
Arithmetic Coding	**Near-opt**	**Yes**	**High**	**Yes (close)**	**JPEG2000, H.264, HEVC**

*Optimal for symbol-by-symbol coding given probabilities

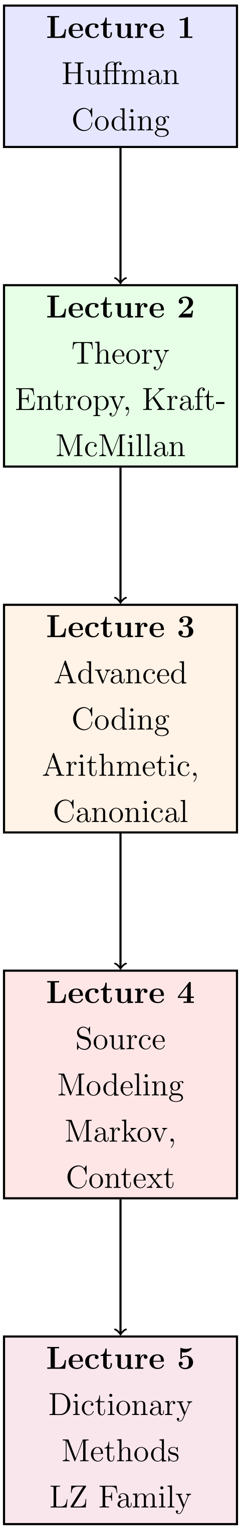
Important

Key Insights:

- **Huffman vs. Arithmetic**: Huffman is simpler but has an ”integer penalty”; Arithmetic approaches entropy bound
- **Modern standards**: Arithmetic coding (CABAC) used in video compression for 10-20% better compression
- **Practical choice**: For general compression, Canonical Huffman (DEFLATE); for media, Arithmetic coding
- **The missing piece**: All these methods assume we have good probability estimates. Where do those come from?

3.9 Forward Look

The Complete Picture: What Comes Next?



Next Lecture: Source Modeling and Statistical Dependence

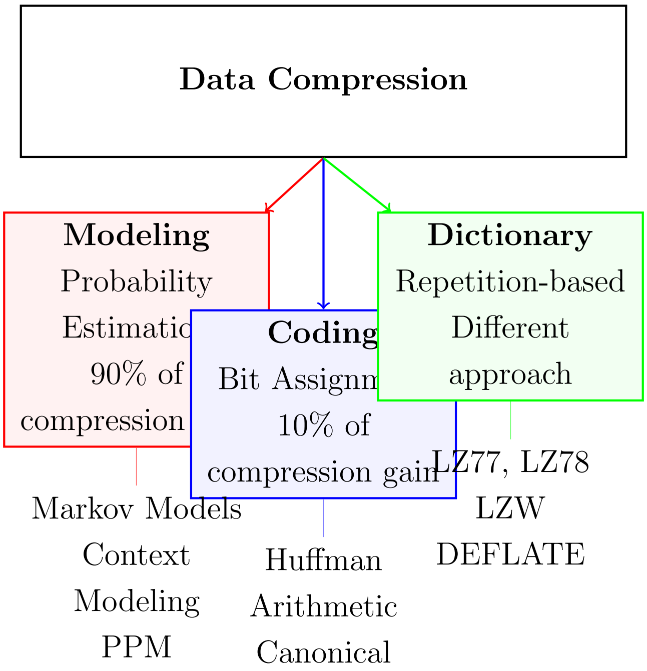
- **The missing half**: We now know how to code efficiently, but where do the probabilities come from?
- **Real data has memory**: 'Q' is usually followed by 'U' in English text
- **Markov models**: Capturing dependencies between symbols

- **Context modeling**: Using past symbols to predict future ones
- **The modeling-coding separation**: Modern compressors separate these two tasks

The Big Question for Next Time:

> **If Arithmetic coding can get within 0.01 bits of entropy,**
> **what's the real limit to compression?**
> **The answer: It's not the coding, it's the *modeling*!**

The Two Pillars of Compression (Revised View):



Exercise 3.0

Exercise 3.1: Given symbols with probabilities: A(0.4), B(0.3), C(0.2), D(0.1)

(a) Construct a Shannon code and compute its expected length

(b) Construct a Shannon-Fano code

(c) Compare with Huffman code from Lecture 2

Exercise 3.1

Exercise 3.2: Convert the following Huffman code to canonical form:

Symbol	Huffman Code
A	0
B	100
C	101
D	110
E	1110
F	1111

Exercise 3.2

Exercise 3.3: Encode the message "ABAC" using arithmetic coding with probabilities: A(0.6), B(0.3), C(0.1). Show each step.

Exercise 3.3

Exercise 3.4: Thinking Ahead: Consider the English phrase "THE QUICK BROWN FOX"

(a) If we use Huffman coding with letter frequencies, what's wrong with this approach?

(b) How might knowing that 'Q' is usually followed by 'U' help compression?

(c) Why would arithmetic coding be better than Huffman for this kind of data?

Exercise 3.4

Exercise: 3.5 Given code lengths: A=2, B=2, C=3, D=3, E=3, F=4

1. Sort the symbols properly
2. Compute the canonical codes
3. Verify no code is a prefix of another
4. What happens if you don't sort by length?

End of Lecture 3 – Advanced Entropy Coding Methods

Next: Lecture 4 – Source Modeling and Statistical Dependence

We now have efficient coding methods. Next: Where do the probabilities come from?

4: Lecture 4: Source Modeling and Statistical Dependence

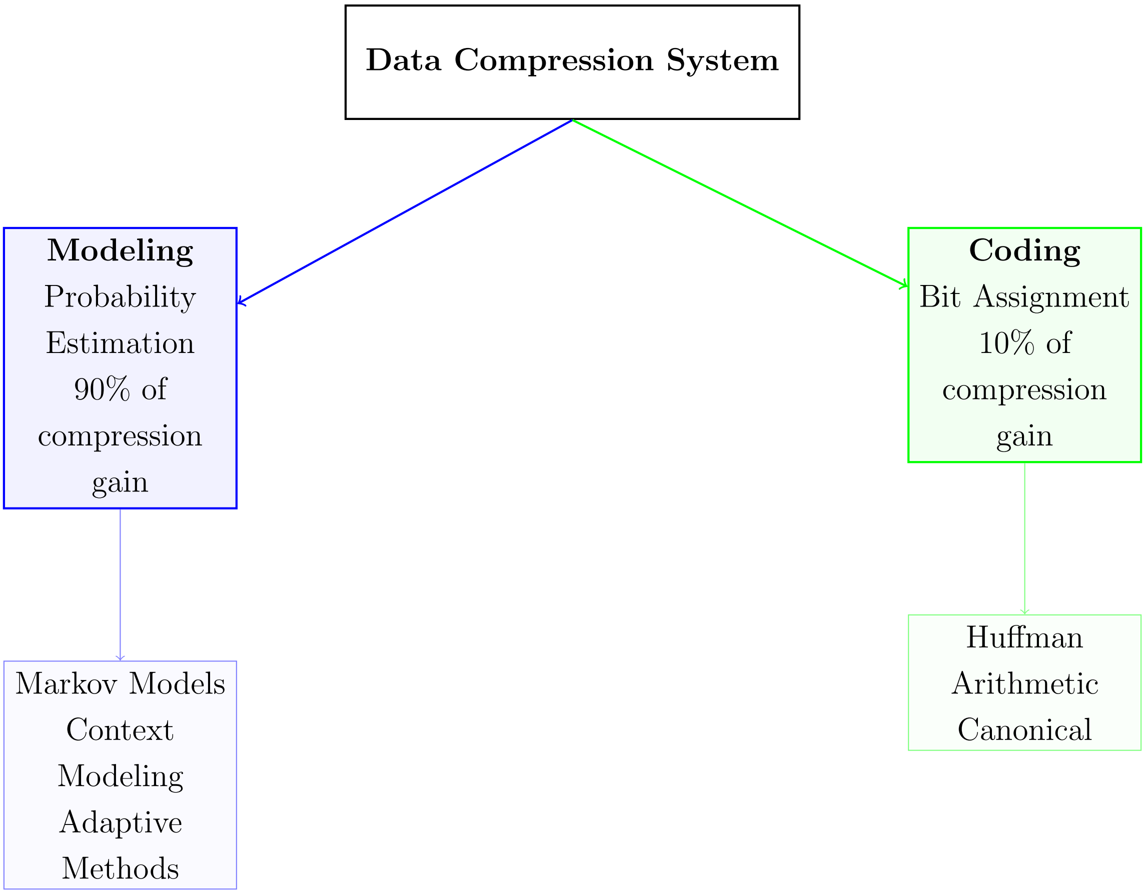
Lecture 4: Beyond Coding – The Power of Source Modeling

4.1 Introduction: Beyond Coding

Important

Key Insight: In the first three lectures, we've focused on **coding** - efficient ways to represent symbols given their probabilities. Now we address the other half: **modeling** - how to get good probability estimates in the first place.

The Big Picture:



Why Real Data Defies IID Assumptions:

- **IID (Independent Identically Distributed)**: Assumption behind simple Huffman
- **Reality**: Data has **memory** and **dependencies**
- Example: In English text, 'Q' is almost always followed by 'U'

- Example: In images, neighboring pixels are highly correlated

Today's Roadmap:

1. Understand statistical dependence in data
2. Learn Markov models for capturing memory
3. Explore context modeling techniques
4. See practical examples with real data
5. Connect modeling to coding (Lecture 3)

4.2 Memoryless vs. Sources with Memory

Definition

Memoryless Source (IID): Each symbol is generated independently of all previous symbols. Probability distribution: $P(X_n = x) = p(x)$ for all n.

Definition

Source with Memory: The probability of a symbol depends on previous symbols. Example: $P(X_n = x | X_{n-1} = y, X_{n-2} = z, \dots)$.

Example

Examples of Dependence in Real Data:

- **Text**: 'TH' is common, 'TQ' is rare
- **Images**: Neighboring pixels have similar colors
- **Audio**: Sound waves have temporal continuity
- **Video**: Consecutive frames are nearly identical
- **Source code**: Keywords, variable names repeat

Important

Measuring Dependence: Autocorrelation

$$\rho(k) = \frac{\mathbb{E}[(X_t - \mu)(X_{t+k} - \mu)]}{\sigma^2}$$

where k is the lag. High $\rho(k)$ means strong dependence at distance k.

4.3 Conditional Entropy and Mutual Information

Definition

Conditional Entropy: The average uncertainty about X given knowledge of Y:

$$H(X|Y) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log_2 p(x|y)$$

Example

Intuition: ”Knowing the Past Helps Predict the Future”
Consider English letters:

- Unconditional: $H(\text{letter}) \approx 4.07$ bits
- Given previous letter: $H(\text{letter}|\text{previous}) \approx 3.36$ bits
- Given previous 2 letters: $H(\text{letter}|\text{previous 2}) \approx 2.77$ bits

Each additional letter of context reduces uncertainty!

Definition

Mutual Information: Measures how much knowing Y reduces uncertainty about X:

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

Example

Worked Example: English Letter Dependence
Let X = current letter, Y = previous letter.

$$\begin{aligned} H(X) &= 4.07 \text{ bits} \\ H(X|Y) &= 3.36 \text{ bits} \\ I(X;Y) &= 4.07 - 3.36 = 0.71 \text{ bits} \end{aligned}$$

This means knowing the previous letter gives us 0.71 bits of information about the current letter.

4.4 Markov Sources

Definition

Markov Property (Memory-m): The future depends only on the last m symbols:

$$P(X_n = x|X_{n-1}, X_{n-2}, \ldots, X_1) = P(X_n = x|X_{n-1}, \ldots, X_{n-m})$$

First-Order Markov Model ($m = 1$):

- Only the immediately previous symbol matters
- Represented by transition probabilities $p_{ij} = P(X_n = j|X_{n-1} = i)$
- Can be shown as a state diagram or transition matrix

Example

Example: Weather Prediction Markov Chain

States: $\{\text{Sunny (S)}, \text{Rainy (R)}\}$

Transition probabilities:

	S	R
S	0.8	0.2
R	0.3	0.7

Interpretation: If today is sunny, 80% chance tomorrow is sunny, 20% chance rainy.

Higher-Order Markov Models:

- Order-k: Depends on last k symbols
- More accurate but exponentially more parameters
- Number of parameters grows as $|\mathcal{A}|^{k+1}$ where $\mathcal{A}$ is alphabet size

Important

Memory-Complexity Trade-off:

- **Order 0**: 26 parameters for English (simple but weak)
- **Order 1**: $26 \times 26 = 676$ parameters
- **Order 2**: $26 \times 26 \times 26 = 17,576$ parameters
- **Order 5**: $26^6 \approx 308$ million parameters!

This is the **context explosion problem**.

4.5 Entropy Rate Revisited

Definition

Entropy Rate of a Stationary Source:
For a stationary stochastic process $\mathcal{X} = (X_1, X_2, \ldots)$, the entropy rate is:

$$H(\mathcal{X}) = \lim_{n\to\infty} \frac{1}{n} H(X_1, X_2, \ldots, X_n)$$

Equivalently (and equal for stationary processes):

$$H(\mathcal{X}) = \lim_{n\to\infty} H(X_n \mid X_1, \ldots, X_{n-1})$$

Special case: Stationary Markov chain of order m
If $\mathcal{X}$ is a stationary m-th order Markov chain, then for all $t > m$:

$$H(X_t \mid X_1, \ldots, X_{t-1}) = H(X_t \mid X_{t-m}, \ldots, X_{t-1})$$

and this conditional entropy is constant over time. Therefore:

$$H(\mathcal{X}) = H(X_{m+1} \mid X_1, \ldots, X_m)$$

No limit needed — the conditional entropy stabilizes immediately.

Example

Example: First-Order Markov Chain ($m = 1$)
Step 1: Define the model Two weather states: S (sunny) and R (rainy).
Transition probabilities (tomorrow's weather depends only on today's):
Let X_t be the weather on day t. Then:

$$\begin{aligned} P(X_{t+1} = S \mid X_t = S) &= 0.8, \qquad & P(X_{t+1} = R \mid X_t = S) &= 0.2 \\ P(X_{t+1} = S \mid X_t = R) &= 0.3, \qquad & P(X_{t+1} = R \mid X_t = R) &= 0.7 \end{aligned}$$

In matrix form (rows = current state, columns = next state):

$$P = \begin{bmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{bmatrix} \qquad \begin{array}{l} \text{Row 1: } X_t = S \\ \text{Row 2: } X_t = R \end{array}$$

Step 2: Find stationary distribution π Stationary distribution $\pi = [\pi_S, \pi_R]$ satisfies $\pi P = \pi$, where $\pi_S = P(X_t = S)$ and $\pi_R = P(X_t = R)$ in the long run.

$$\begin{cases} \pi_S = 0.8\pi_S + 0.3\pi_R \\ \pi_R = 0.2\pi_S + 0.7\pi_R \\ \pi_S + \pi_R = 1 \end{cases}$$

Solving (from first equation):

$$\pi_S - 0.8\pi_S = 0.3\pi_R \Rightarrow 0.2\pi_S = 0.3\pi_R \Rightarrow \pi_S = 1.5\pi_R$$

$$1.5\pi_R + \pi_R = 1 \Rightarrow 2.5\pi_R = 1 \Rightarrow \pi_R = 0.4$$

$$\pi_S = 1.5 \times 0.4 = 0.6$$

$$\pi = [\pi_S, \pi_R] = [0.6, 0.4]$$

Step 3: Compute conditional entropies *Uncertainty about tomorrow given today's weather:*
Given $X_t = S$: Tomorrow's distribution is $[0.8, 0.2]$

$$H(X_{t+1} \mid X_t = S) = -0.8 \log_2 0.8 - 0.2 \log_2 0.2 \approx 0.7219 \text{ bits}$$

Given $X_t = R$: Tomorrow's distribution is $[0.3, 0.7]$

$$H(X_{t+1} \mid X_t = R) = -0.3 \log_2 0.3 - 0.7 \log_2 0.7 \approx 0.8813 \text{ bits}$$

Step 4: Compute entropy rate For a stationary first-order Markov chain:

$$H(\mathcal{X}) = H(X_{t+1} \mid X_t) = \sum_{s \in \{S,R\}} P(X_t = s) \cdot H(X_{t+1} \mid X_t = s)$$

With $P(X_t = s) = \pi_s$ (stationarity):

$$H(\mathcal{X}) = \pi_S \cdot H(X_{t+1} \mid X_t = S) + \pi_R \cdot H(X_{t+1} \mid X_t = R)$$

$$= 0.6 \times 0.7219 + 0.4 \times 0.8813 = 0.43314 + 0.35252 = 0.78566 \text{ bits}$$

$$\approx 0.786 \text{ bits per day}$$

Important

What This Means for Data Compression:
Two approaches to compression:

1. **IID (Independent Identically Distributed) assumption:**

- Treat each symbol as independent of previous ones
- Best achievable rate: $H(X)$ (marginal entropy)
- Example: English letters ≈ 4.07 bits/letter

2. **With modeling (using context):**
   - Account for dependencies between symbols (Markov structure)
   - Use conditional probabilities based on previous symbols
   - Best achievable rate: $H(\mathcal{X})$ (entropy rate)
   - Example: English text ≈ 2.3 bits/letter

Compression gain:

$$\frac{H(X) - H(\mathcal{X})}{H(X)} \approx \frac{4.07 - 2.3}{4.07} \approx 43\%$$

Up to $\sim 45\%$ better compression by exploiting dependencies!

4.6 Context Modeling in Practice

Definition

Context Modeling: Maintain separate probability distributions for each possible context (history).

Fixed-Length vs. Variable-Length Contexts:

- **Fixed-length**: Always use last k symbols as context
- **Variable-length**: Use longest matching context in database
- Example: PPM (Prediction by Partial Matching) uses variable-length

Example

The Context Explosion Problem:

For English text (26 letters + space):

Order	Contexts	Parameters
0	1	27
1	27	729
2	729	19,683
3	19,683	531,441
4	531,441	14,348,907
5	14,348,907	387,420,489

By order 5: 387 million parameters need estimation!

Solutions to Context Explosion:

1. **Escaping**: Fall back to lower-order model when context unseen
2. **Blending**: Combine predictions from different order models
3. **Pruning**: Remove low-frequency contexts
4. **Adaptive methods**: Update probabilities as data arrives

4.7 Case Study: Text Compression Modeling

Example

1. Entropy Estimates for English Text (Theoretical Limits)

Model/Estimation Method	**Bits/Letter**	**Year/Source**
Letter frequencies only (Order-0)	4.07	Shannon 1948
+ 1st order Markov (bigrams)	3.36	
+ 2nd order (trigrams)	2.77	
+ 3rd order	2.43	
Human prediction experiment	1.3	Shannon 1951
Modern best estimates	1.0–1.5	Various studies

2. Practical Compression Performance

Compressor/Method	**Bits/Letter**	**Notes**
gzip (LZ77 + Huffman)	2.5–3.0	Fast, widely used
bzip2 (BWT + MTF + Huffman)	2.3–2.8	Better but slower
PPMd	2.1–2.4	Context modeling
PAQ variants	1.5–1.8	State of the art, very slow

Key insight: Better models (using more context) → lower entropy → better compression.

But practical compressors face tradeoffs: memory, speed, and model complexity.

PPM (Prediction by Partial Matching):

- Uses **variable-length** contexts
- Tries highest-order model first
- Escapes to lower order if context unseen
- Blends probabilities from different orders
- State-of-the-art for text compression in 1990s

Important

Practical Entropy Reduction:

$$\begin{aligned} \text{IID model (Huffman)} &: 4.07 \text{ bits/letter} \\ \text{With context modeling} &: 2.23 \text{ bits/letter} \\ \text{Improvement} &: 45\% \text{ better compression!} \end{aligned}$$

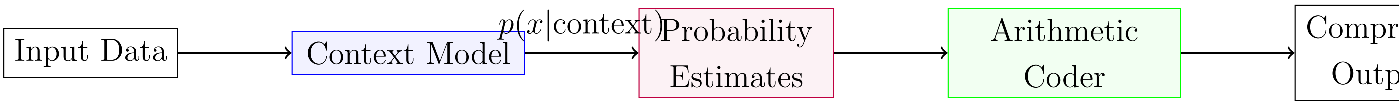
4.8 The Modeling–Coding Separation Principle

Definition

Modeling–Coding Separation: Modern compressors separate probability estimation (modeling) from bit assignment (coding).

Historical Evolution:

- **Early**: Integrated (Huffman builds tree from frequencies)
- **Modern**: Separated (Model → Probabilities → Arithmetic Coder)



Example

How PPM + Arithmetic Beats Huffman:

1. **PPM**: Sees context ”TH” → predicts E with 80% probability
2. **Arithmetic**: Encodes E using $p = 0.8 \rightarrow$ 0.32 bits
3. **Huffman**: Would need at least 1 bit for any symbol

4. **Gain**: 0.32 bits vs 1+ bits = 3× better for this symbol!

Important

Why Arithmetic Coding is the Perfect Backend:

- Can handle **fractional bits** per symbol
- Accepts **changing probabilities** symbol by symbol
- Works with **adaptive models** naturally
- Achieves entropy bound for good models

4.9 Adaptive vs. Static Modeling

Definition

Static Models: Train once on representative data, use fixed model for all files.

- **Pros**: Fast encoding/decoding
- **Cons**: Model may not match specific file
- **Example**: Early text compressors using English statistics

Definition

Semi-Adaptive (Two-Pass): First pass: collect statistics; Second pass: encode.

- **Pros**: Tailored to specific file
- **Cons**: Need to transmit model (overhead)
- **Example**: Standard Huffman with tree transmission

Definition

Fully Adaptive (One-Pass): Update model while encoding/decoding.

- **Pros**: No model transmission, adapts to local changes
- **Cons**: Slower, initial poor compression
- **Example**: Adaptive Huffman, PPM with update

Example

Comparison in Practice:

Type	Compression	Speed	Memory
Static	Medium	Fast	Low
Semi-Adaptive	Good	Medium	Medium
Fully Adaptive	Best	Slow	High

Choice depends on application constraints!

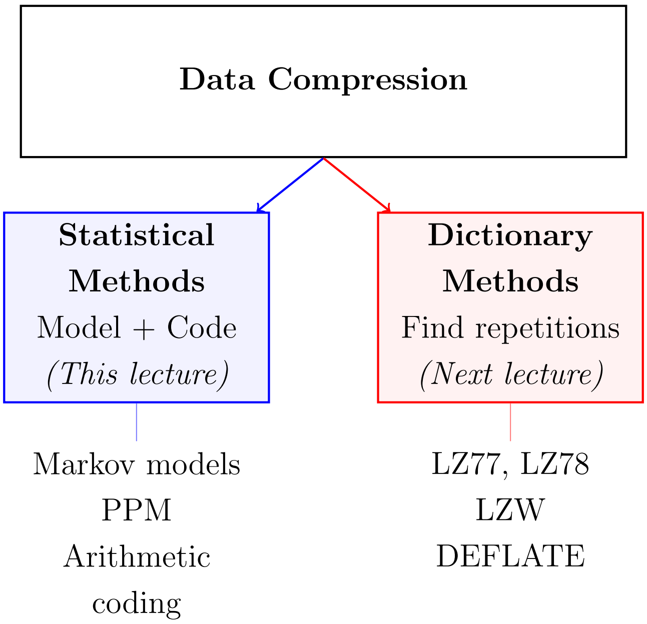
4.10 Summary and Forward Look

Important

Key Takeaways:

1. **The real compression is in modeling**, not just coding
2. **Context matters**: Using past symbols reduces uncertainty
3. **Markov models** capture memory in data
4. **Context explosion** limits practical model order
5. **Arithmetic coding** enables efficient use of good models
6. **Adaptive methods** avoid model transmission overhead

The Two Pillars Revisited:



Data Compression

Statistical Methods
Model + Code
(This lecture)

Dictionary Methods
Find repetitions
(Next lecture)

Markov models
PPM
Arithmetic coding

LZ77, LZ78
LZW
DEFLATE

Preview: Next Lecture on LZ Family (Dictionary Methods):

- A completely different approach: find repeating patterns
- No probability estimation needed
- Works well for files with exact repetitions
- Used in ZIP, GIF, PDF, and many modern formats
- Often combined with statistical methods in practice

Exercise 4.0

Exercise 4.1: Given the first-order Markov chain for weather:

	S	R
S	0.7	0.3
R	0.4	0.6

a) Find the stationary distribution $\pi = [\pi_S, \pi_R]$ b) Calculate the entropy rate $H(\mathcal{X})$ c) How does this compare to an IID source with $P(S) = 0.55, P(R) = 0.45$?

Exercise 4.1

Exercise 4.2: Consider the text fragment: "THE CAT SAT ON THE MAT" a) Build an order-1 (bigram) model for this text b) Calculate $H(X)$ (order-0 entropy) c) Calculate $H(X|Y)$ where Y is previous letter (order-1 conditional entropy) d) How much mutual information exists between consecutive letters?

Exercise 4.2

Exercise 4.3: Explain why arithmetic coding is better than Huffman coding when used with: a) A high-order Markov model b) An adaptive context model c) A model that gives very skewed probabilities (e.g., $p = 0.99$ for one symbol)

End of Lecture 4 – Source Modeling and Statistical Dependence

Next: Dictionary-based Compression (LZ Family)

5: Lecture 5: Dictionary-Based Compression: The Lempel–Ziv Revolution

5.1 Motivation: Hitting the Limits of Statistical Coding

Statistical coding methods such as Huffman and arithmetic coding achieve near-optimal compression *when the source statistics are known or can be accurately estimated.* However, these methods fundamentally rely on estimating probabilities over symbols or blocks of symbols.

As discussed earlier, block-based statistical coding faces a fundamental dilemma.

5.1.1 Recap: The Block Coding Dilemma

Increasing block length allows a coder to better capture dependencies between symbols and approach the entropy rate of the source. However, this comes at a steep cost:

- The number of possible blocks grows exponentially with block length.
- Estimating probabilities reliably requires exponentially more data.
- Memory and computational complexity quickly become impractical.

As a result, practical statistical coders are constrained to small contexts and local dependencies.

5.1.2 The Promise of Exploiting Long-Range Repetition

Real-world data—text, executable files, logs, DNA sequences—often contains *long-range repetition.* Patterns may repeat far apart, well beyond the reach of fixed-size statistical contexts.

Important

Statistical coding models *how often* symbols occur, but does not directly model *where long repeated patterns occur.*

This observation motivates a radically different approach to compression.

5.2 Paradigm Shift: From Statistics to Dictionaries

Instead of estimating probabilities, dictionary-based compression learns the source *by example.*

5.2.1 Core Philosophy of Dictionary Coding

Dictionary-based coders operate on a simple idea:

> *Replace repeated substrings by references to earlier occurrences.*

As the input is processed sequentially, a dictionary of previously seen substrings is constructed. Future occurrences are encoded by references into this dictionary.

Important

You can think of Lempel–Ziv methods as *learning the source structure rather than estimating probabilities.*

Crucially, the encoder and decoder process the input in exactly the same order and therefore build identical dictionaries *without transmitting the dictionary explicitly.*

5.2.2 Explicit vs. Implicit (Adaptive) Dictionaries

Dictionary-based methods fall into two categories:

- **Implicit dictionaries**: The dictionary is defined by a sliding window of recent output (e.g., LZ77, LZSS).
- **Explicit dictionaries**: The dictionary is stored and indexed explicitly (e.g., LZ78).

We now study the foundational Lempel–Ziv algorithms that underpin modern compression.

5.3 LZ77: The Sliding Window Algorithm

Published in 1977 by Abraham Lempel and Jacob Ziv, LZ77 introduced the concept of using recent history as a dictionary.

5.3.1 The Search Buffer and Look-Ahead Buffer

LZ77 maintains a sliding window divided into two parts:

- **Search buffer**: Contains the N most recently processed symbols.
- **Look-ahead buffer**: Contains the next L symbols to be encoded.

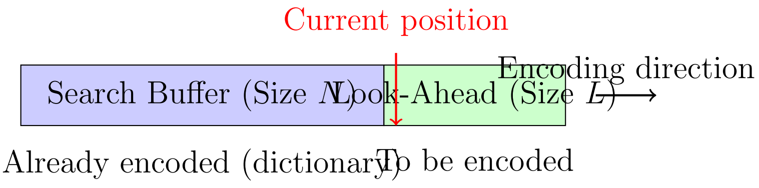


Figure 1: LZ77 sliding window structure

5.3.2 Encoding Tuples: (Offset, Length, Next Symbol)

At each step, the encoder finds the longest match in the search buffer for the prefix of the look-ahead buffer. The match is encoded as:

$$(\text{offset}, \text{length}, \text{next symbol})$$

where:

- **Offset**: Distance backward to the match (1-based indexing).
- **Length**: Number of matching symbols.
- **Next symbol**: The first symbol after the match (or EOF).

If no match is found, the encoder outputs $(0, 0, c)$ where c is the literal symbol from the look-ahead buffer.

5.3.3 Step-by-Step Encoding Example

Consider encoding the string `abracadabra` with a sufficiently large window:

Step	Search Buffer	Look-Ahead	Output
1	–	`abracadabra`	$(0, 0, \texttt{a})$
2	`a`	`bracadabra`	$(0, 0, \texttt{b})$
3	`ab`	`racadabra`	$(0, 0, \texttt{r})$
4	`abr`	`acadabra`	$(3, 1, \texttt{c})$
5	`abrac`	`adabra`	$(5, 1, \texttt{d})$
6	`abraca`	`abra`	$(7, 4, \$)$

Where $ denotes end-of-file. Notice how the second occurrence of ”`abra`” (length 4) is encoded by referencing back to offset 7 (the first occurrence starting at position 1 in our 0-indexed string).

5.3.4 Decoding Process: Simple Reconstruction

Decoding is straightforward and proceeds sequentially:

Algorithm 2 LZ77 Decoding

```
procedure DECODE($stream$)
    Initialize output ← empty string
    for each tuple $(o, l, c)$ in $stream$ do
        if $o = 0$ and $l = 0$ then
            Append $c$ to output
        else
            $start \leftarrow \text{length}(output) - o$
            for $i \leftarrow 0$ to $l - 1$ do
                Append output[$start + i$] to output
            end for
            if $c \neq \$$ then
                Append $c$ to output
            end if
        end if
    end for
    return output
end procedure
```

Example

Decoding Example:
Decoding the sequence from our example:

$$(0, 0, \texttt{a}) \to \texttt{a}$$
$$(0, 0, \texttt{b}) \to \texttt{ab}$$
$$(0, 0, \texttt{r}) \to \texttt{abr}$$
$$(3, 1, \texttt{c}) \to \texttt{abrac}$$
$$(5, 1, \texttt{d}) \to \texttt{abracad}$$
$$(7, 4, \$) \to \texttt{abracadabra}$$

Important

Why Decoding Works: Every referenced symbol has already been reconstructed. The decoder maintains exactly the same output buffer that served as the encoder's search buffer.

5.3.5 Design Parameters: Window Size and Match Limits

LZ77 has two key parameters:

- **Search buffer size (N)**: Larger buffers find more matches but increase memory and search time.
- **Maximum match length (L)**: Limits how far ahead the encoder can look.

Practical implementations use efficient data structures (hash tables, suffix arrays) to find matches in $O(1)$ or $O(\log N)$ time rather than the naive $O(NL)$.

5.4 LZSS: Improving Practical Efficiency

Published in 1982 by James Storer and Thomas Szymanski, LZSS addresses a key inefficiency in LZ77.

5.4.1 The Problem with LZ77's Encoding

LZ77's (o, l, c) format has two inefficiencies:

1. **No-match penalty**: $(0, 0, c)$ uses more bits than the literal c alone.
2. **Short-match penalty**: Encoding a match of length 2 might not save bits after accounting for tuple overhead.

5.4.2 Flag Bits: Literal vs. Match

LZSS introduces a **flag bit** preceding each encoded element:

- **Flag = 0**: Next item is a literal symbol (8 bits for ASCII).
- **Flag = 1**: Next item is a match pair (offset, length).

Important

Key Insight: LZSS separates the *parsing decision* ("literal or match?") from the *data encoding* itself. This conceptual separation is crucial for understanding modern hybrid compressors.

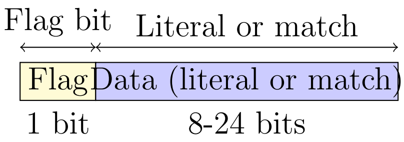


Figure 2: LZSS encoding format

5.4.3 Match Threshold

LZSS introduces a **match threshold**: only encode matches that are *long enough* to actually save bits. Typically:

$$\text{Encode match if: } l \times \text{bits\_per\_symbol} > \text{bits\_for\_flag} + \text{bits\_for\_offset} + \text{bits\_for\_length}$$

For typical parameters (8-bit symbols, 12-bit offset, 4-bit length):

$$\text{Match if: } l \times 8 > 1 + 12 + 4 = 17 \implies l \geq 3$$

So LZSS would only encode matches of length 3 or more.

5.4.4 LZSS Encoding Example

Let's encode `abracadabra` with LZSS (match threshold = 3):

Step	Look-Ahead	Action	Output
1	`abracadabra`	No match ≥ 3	Flag=0, `a`
2	`bracadabra`	No match ≥ 3	Flag=0, `b`
3	`racadabra`	No match ≥ 3	Flag=0, `r`
4	`acadabra`	Match "`a`" (length=1)	Flag=0, `a`
5	`cadabra`	No match ≥ 3	Flag=0, `c`
6	`adabra`	Match "`a`" (length=1)	Flag=0, `a`
7	`dabra`	No match ≥ 3	Flag=0, `d`
8	`abra`	Match "`abra`" (length=4)	Flag=1, (7,4)

Output in binary (simplified):

- Flag bits: 0 0 0 0 0 0 0 1
- Literals: a b r a c a d
- Match: offset=7, length=4

5.4.5 Why LZSS Matters

Important

LZSS is the foundation for the **DEFLATE** algorithm used in ZIP files, gzip, PNG images, and HTTP compression. Its innovations—flag bits and match thresholds—make dictionary compression practical for real-world use.

5.5 LZ78: The Dictionary Growth Algorithm

Published in 1978, LZ78 takes a different approach: it builds an explicit dictionary that grows as encoding proceeds.

5.5.1 Building an Explicit Dictionary from Scratch

Unlike LZ77's sliding window, LZ78 builds a dictionary of phrases:

- Starts with empty dictionary (index 0 represents empty string).
- Each new phrase is added to the dictionary.
- Dictionary indices are assigned sequentially.

5.5.2 Encoding Pairs: (Dictionary Index, New Symbol)

LZ78 outputs pairs:

$$(\text{index}, \text{symbol})$$

where:

- **Index**: Dictionary index of the longest matching prefix.
- **Symbol**: The next symbol after the match.

The new phrase (dictionary[index] + symbol) is added to the dictionary.

Algorithm 3 LZ78 Encoding

```
procedure ENCODE($input$)
    Initialize dict $\leftarrow \{\epsilon\}$ (empty string at index 0)
    $current \leftarrow \epsilon$
    for each symbol $s$ in $input$ do
        if $current + s$ is in dict then
            $current \leftarrow current + s$
        else
            $i \leftarrow$ index of $current$ in dict
            Output $(i, s)$
            Add $current + s$ to dict
            $current \leftarrow \epsilon$
        end if
    end for
    if $current \neq \epsilon$ then
        Output $(index(current), \$)$
    end if
end procedure
```

5.5.3 Worked Example: From String to Codes

Encoding `abracadabra`:

Step	Current Phrase	Output	New Dictionary Entry
1	`a`	$(0, \mathtt{a})$	1: `a`
2	`b`	$(0, \mathtt{b})$	2: `b`
3	`r`	$(0, \mathtt{r})$	3: `r`
4	`ac`	$(1, \mathtt{c})$	4: `ac`
5	`ad`	$(1, \mathtt{d})$	5: `ad`
6	`abr`	$(1, \mathtt{b})$	6: `ab`

The output sequence: $(0, a)$, $(0, b)$, $(0, r)$, $(1, c)$, $(1, d)$, $(1, b)$.

5.5.4 Comparison with LZ77/LZSS

Important

LZ78 vs LZ77/LZSS:

- **LZ77/LZSS**: Fast adaptation, limited memory (window), good for local repeats.
- **LZ78**: Slow start, unlimited growth, captures long-term repetitions.
- In practice, LZ77/LZSS variants dominate due to better memory behavior and faster adaptation.

5.6 Theoretical Properties: Why Lempel–Ziv Works

5.6.1 The Universality Principle

Lempel–Ziv algorithms have a remarkable theoretical property:

Definition

Universal Compression: An algorithm is universal if it can compress any stationary ergodic source to its entropy rate *without prior knowledge* of the source statistics.

Both LZ77 and LZ78 are universal compressors.

5.6.2 Asymptotic Optimality

Theorem 5.1 (Ziv & Lempel, 1977-1978)**.** *For any stationary ergodic source with entropy rate H, let L_n be the length of the LZ77 or LZ78 encoding of n source symbols. Then:*

$$\lim_{n\to\infty} \frac{\mathbb{E}[L_n]}{n} = H$$

Intuition. As n grows large:

- The algorithm sees enough repetitions to build an accurate implicit model.
- Common phrases get short codes (through references).
- Rare phrases get longer codes (literal transmission).
- The average code length approaches the self-information of phrases.

□

5.6.3 Practical Implications of Universality

Important

Why Universality Matters:

- No need to estimate probabilities or build explicit models.
- Works on any data type (text, images, executables, DNA).
- Adapts automatically to source characteristics.
- Explains why LZ methods work so well in practice.

5.7 Bridging Paradigms: Dictionary vs. Entropy Coding

5.7.1 Complementary Approaches

Dictionary coding and entropy coding address different aspects of compression:

- **Dictionary coding**: Exploits *structural redundancy* (repetitions).
- **Entropy coding**: Exploits *statistical redundancy* (skewed symbol frequencies).

5.7.2 The Hybrid Approach

Modern compressors combine both approaches:

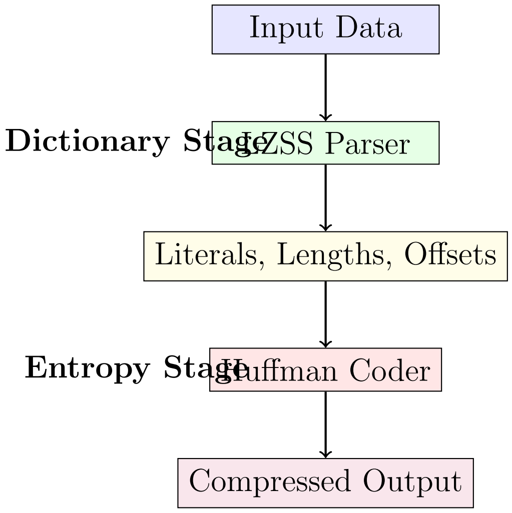


Figure 3: Hybrid compression pipeline (e.g., DEFLATE algorithm)

The DEFLATE algorithm (used in ZIP, PNG, gzip) works precisely this way:

1. LZSS finds repeated strings, outputs literals and matches.
2. Huffman coding compresses:
   - Literal symbols (0-255)
   - Match lengths (257-285)
   - Match distance codes (0-29)

5.8 Summary and Forward Look

5.8.1 Key Takeaways

- **Paradigm shift**: Dictionary coding exploits repetition rather than probabilities.
- **LZ77**: Sliding window approach, encodes matches as triples.
- **LZSS**: Practical improvement with flag bits and match thresholds.
- **LZ78**: Explicit dictionary growth, encodes pairs.
- **Universality**: LZ methods work on any stationary source without prior knowledge.
- **Modern practice**: Hybrid systems (LZSS + Huffman) dominate.

5.8.2 The Road Ahead: DEFLATE and Beyond

In our next lecture, we'll study the DEFLATE algorithm in detail:

- How LZSS and Huffman coding are combined.

- The gzip/ZIP/PNG compression pipeline.
- Practical implementation considerations.
- Performance trade-offs and optimizations.

Important

Historical Context: The Lempel–Ziv papers (1977, 1978) are among the most cited in information theory. Their algorithms underpin much of today's digital compression infrastructure. Understanding these foundations is essential for working with modern compression standards.

End of Chapter Exercises

Exercise 5.0

Exercise 5.1 (LZ77 Encoding)
Encode the string `TOBEORNOTTOBEORTOBEORNOT` using LZ77 with a search buffer of size 12. Show your work step by step with the search buffer, look-ahead buffer, and output tuples.

Exercise 5.1

Exercise 5.2 (LZSS Efficiency Analysis)
Compare LZ77 and LZSS for encoding a sequence with parameters: 8-bit symbols, 12-bit offsets, 4-bit lengths.

(a) How many bits does LZ77 use for a non-match?

(b) How many bits does LZSS use for a literal?

(c) What is the minimum match length that saves bits in LZSS?

(d) When would LZ77 be more efficient than LZSS?

Exercise 5.2

Exercise 5.3 (LZ78 Dictionary Growth)
Encode the string `ABABABABA` using LZ78. Show the dictionary after each step and the complete output sequence. How many dictionary entries are created?

Exercise 5.3

Exercise 5.4 (Algorithm Comparison)
For each string below, predict which would perform better: LZ77/LZSS or LZ78? Explain why.

(a) `AAAAAAAAAAAAAAAAAAAA` (20 A's)

(b) `ABCDEFGHIJKLMNOPQRSTUVWXYZ`

(c) `ABABABABABABABABABAB`

(d) A file containing the same 1KB block repeated 100 times

Exercise 5.4

Exercise 5.5 (Decoding Challenge)
Decode the following LZSS encoded sequence (match threshold = 3):

```
0 T
0 H
0 E
1 (3,5)
0 _
0 _
1 (5,4)
0 .
```

Where: - Each line represents one encoded element - Flag 0 indicates a literal character - Flag 1 indicates a match (offset, length) - _ represents space character - . represents period
What is the original string? Show your decoding step by step.

Exercise 5.5

Exercise 5.6 (Window Size Analysis)
Consider LZ77 with search buffer size N.

(a) How many bits are needed to encode an offset value?

(b) If we increase N from 4096 to 65536, how many more bits are needed for offsets?

(c) What is the trade-off in choosing N?

(d) In practice, why might we choose $N = 32768$ over $N = 65536$ even if memory is available?

Exercise 5.6

Exercise 5.7 (Universal Compression Intuition)
Explain in your own words why Lempel–Ziv algorithms are universal. Why don't they need to know source statistics in advance? How do they "learn" the source

structure?

Exercise 5.7

Exercise 5.8 (Hybrid Coding Design)
Design a simple hybrid compressor that:

(a) Uses LZSS with match threshold 3

(b) Collects statistics on literals, lengths, and offsets

(c) Applies Huffman coding to each alphabet separately

How would the decoder know which Huffman trees to use?

Exercise 5.8

Exercise 5.9 (Search Efficiency)
A naive LZ77 implementation searches for matches by comparing the look-ahead buffer against every position in the search buffer.

(a) What is the time complexity of this approach?

(b) Describe how a hash table can improve search efficiency.

(c) What information would you store in the hash table?

(d) How would you handle hash collisions?

Exercise 5.9

Exercise 5.10 (Theoretical Limits)
Prove or provide intuition for:

(a) LZ77 cannot achieve compression ratio better than the entropy rate for i.i.d. sources.

(b) LZ78 may perform poorly on very short inputs.

(c) For sources with long-range dependencies, LZ methods can outperform block Huffman coding even with large blocks.

6: Lecture 6: DEFLATE Algorithm: The Standard in Practice

6.1 Introduction: The Universal Compression Standard

The DEFLATE compression algorithm, specified in RFC 1951 (1996), is arguably the most widely deployed compression algorithm in history. Its success lies not in being the theoretically optimal algorithm, but in striking an exceptional balance between compression ratio, speed, and implementation simplicity.

Definition

DEFLATE: A lossless data compression algorithm combining LZ77-style string matching with Huffman coding. It is the compression engine behind gzip, ZIP, PNG, and HTTP compression.

6.1.1 Where DEFLATE Is Used: gzip, ZIP, PNG, HTTP

- **gzip**: The Unix compression utility (RFC 1952)
- **ZIP file format**: Phil Katz's PKZIP format (1989), later standardized
- **PNG image format**: Replaced GIF, uses DEFLATE on filtered scanlines
- **HTTP compression**: Content-Encoding: gzip/deflate in web traffic
- **PDF documents**: FlateDecode filter for stream compression
- **Java JAR files**: Based on ZIP format

6.1.2 Historical Context: Phil Katz, ZIP Format, and Open Standards

The story of DEFLATE is intertwined with the "compression wars" of the late 1980s-1990s:

- **1985**: Terry Welch's LZW algorithm patented (used in GIF)
- **1986**: Phil Katz creates PKARC (based on ARC format)
- **1989**: Katz releases PKZIP with proprietary compression
- **1991**: gzip created as free alternative to UNIX compress (which used LZW)
- **1993**: Unisys enforces LZW patents, creating demand for patent-free alternatives
- **1996**: DEFLATE specification published as RFC 1951

The key innovation was making DEFLATE specification **open** and **royalty-free**, leading to widespread adoption.

6.2 High-Level Architecture: A Two-Stage Pipeline

DEFLATE employs a classic transform-coding approach:

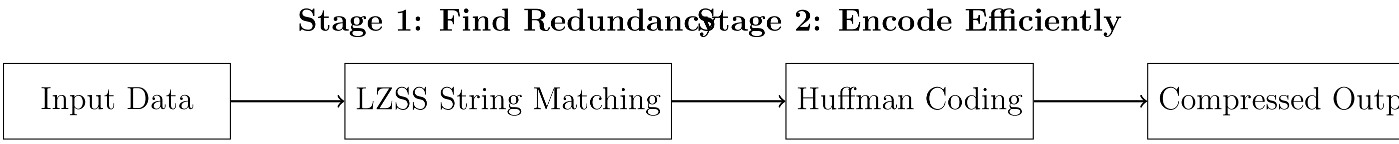


Figure 4: DEFLATE's two-stage compression pipeline

6.2.1 Stage 1: LZSS String Matching (Finding Redundancy)

The LZSS (Lempel-Ziv-Storer-Szymanski) stage identifies repeated sequences in the data:

- Scans input with a **sliding window** (32KB history buffer)
- For each position, looks for longest match in recent history
- Outputs either:
  - **Literal byte**: When no good match found
  - **(Length, Distance) pair**: When match found

Example: For the string "the cat sat on the mat":

- First occurrence of "the": output as literals 't','h','e'
- Second occurrence of "the": output as (length=3, distance=15)

6.2.2 Stage 2: Huffman Coding (Encoding the Reduced Data)

The Huffman stage compresses the output from Stage 1:

- Takes stream of literals and (length,distance) pairs
- Builds **two Huffman trees**:
  - One for literals and lengths (combined alphabet)
  - One for distances
- Uses **canonical Huffman codes** for compact representation

6.2.3 Why the Hybrid Approach Works: Transforming Redundancy

The genius of DEFLATE is in how it **transforms redundancy**:

1. **LZSS finds structural redundancy**: Repeated sequences become (length,distance) pairs
2. **Huffman finds statistical redundancy**: Frequent symbols get shorter codes
3. Together they exploit **both types** of redundancy in data

Important

Key Insight: LZSS is excellent at finding *long-range* repetition (phrases, patterns). Huffman is excellent at compressing *skewed distributions.* Together they handle both English text (skewed letter frequencies + repeated words) effectively.

6.2.4 Design Philosophy: Fast Decoding Over Maximum Compression

DEFLATE was designed with clear priorities:

Priority	**Implementation Choice**
Fast decoding	Canonical Huffman (O(1) decode)
Memory efficiency	32KB sliding window (tiny by today's standards)
Implementation simplicity	Two independent stages
Streaming capability	Independent compression blocks
Acceptable compression	"Good enough" ratio for most data

This "fast decode" philosophy made DEFLATE ideal for distribution formats (ZIP, PNG) where files are compressed once but decompressed many times.

6.3 LZSS in DEFLATE: Detailed Implementation

6.3.1 Sliding Window: 32KB History and Lookahead Buffer

DEFLATE's LZSS uses specific, fixed parameters:

- **History buffer**: 32,768 bytes (2^{15}) of recently processed data
- **Lookahead buffer**: 258 bytes maximum match length
- **Minimum match length**: 3 bytes (shorter matches not worth encoding)

Example

Why 32KB?

- 32KB was a practical limit in 1990s memory-constrained systems
- Larger windows provide diminishing returns for most data
- Cache-friendly size for modern processors
- Still adequate for most short-to-medium range repetitions

6.3.2 Hash Chains for Fast Match Finding

Instead of brute-force searching 32KB for matches, DEFLATE uses a hash table:

Algorithm 4 DEFLATE Match Finding with Hash Chains

```
procedure FindBestMatch($current\_position$, $data$)
    $hash \leftarrow$ hash($data[current\_position : current\_position + 3]$)
    $best\_length \leftarrow 0$
    $best\_distance \leftarrow 0$
    for each $prev\_pos$ in $hash\_table[hash]$ do
        if $current\_pos - prev\_pos > 32768$ then        ▷ Outside window
            continue
        end if
        $length \leftarrow$ MatchLength($prev\_pos, current\_pos$)
        if $length > best\_length$ then
            $best\_length \leftarrow length$
            $best\_distance \leftarrow current\_pos - prev\_pos$
        end if
    end for
    return ($best\_length, best\_distance$)
end procedure
```

The hash is typically computed from the first 3 bytes of the potential match. Hash chains link all positions with the same 3-byte prefix.

6.3.3 Lazy Matching and Greedy Heuristics

DEFLATE uses sophisticated heuristics to find better matches:

- **Greedy parsing**: Take the longest match at current position
- **Lazy evaluation**: Check if skipping 1 byte finds a better match

- **Lazy matching**: Actually implement lazy evaluation

Example

Lazy Matching Example: Data: "abcdeabcdfabcde"

1. At position 0: "abcde" matches at position 5 (length=5)
2. **Greedy**: Encode (length=5, distance=5) immediately
3. **Lazy**: Check position 1: "bcdeabcdfabcde" has match at position 6 (length=13)
4. Better! So output 'a' as literal, then encode long match

Lazy matching finds longer matches but requires more computation.

6.3.4 Output Format: (Length, Distance) Pairs vs. Literal Bytes

The LZSS stage produces a stream of symbols from two alphabets:

Type	**Range**	**Encoding Details**
Literal bytes	0–255	Direct byte values
Length codes	3–258	Base code + 0–5 extra bits for longer lengths
Distance codes	1–32768	Base code + 0–13 extra bits in logarithmic ranges
End-of-block	256	Special marker (end of DEFLATE block)

Important encoding details:

- **Lengths 3–258**: Encoded using symbols 257–285
  - For example: Length 3 = symbol 257 (no extra bits)
  - Length 10 = symbol 264 (no extra bits)
  - Length 258 = symbol 285 (no extra bits)
  - Lengths 11–258 use 1–5 extra bits for precise encoding
- **Distances 1–32768**: Encoded using symbols 0–29 with logarithmic ranges
  - Example ranges:
    - Symbol 0: Distance 1 (no extra bits)
    - Symbol 4: Distances 5–6 (1 extra bit)
    - Symbol 29: Distances 24577–32768 (13 extra bits)
  - Distance 100 would use symbol for range 97–128 with extra bits

Example

Encoding Distance 100:

1. Distance 100 falls in range 97–128
2. According to DEFLATE specification:
    - Range 97–128 corresponds to distance symbol 18
    - Base distance for this symbol: 97
    - Extra bits needed: 2 bits (since 128-97+1 = 32 possibilities)
3. Compute extra bits: 100 - 97 = 3
4. Encode as: symbol 18 followed by binary '11' (3 in 2 bits)

This logarithmic encoding allows representing 32768 possible distances with only 30 symbols plus extra bits, significantly reducing the Huffman alphabet size.

6.4 DEFLATE's Unique Huffman Coding Scheme

This is where DEFLATE shows its brilliance. Instead of storing full Huffman trees, it stores only **code lengths**.

6.4.1 Two Alphabets: Literals (0-285) and Distances (0-29)

DEFLATE uses separate Huffman trees for different symbol types:

- **Literals/Lengths alphabet**: 286 symbols
    - 0-255: Literal bytes
    - 256: End-of-block marker
    - 257-285: Length codes (with extra bits)
- **Distances alphabet**: 30 symbols
    - 0-3: Direct distances 1-4
    - 4-29: Distance codes with extra bits

6.4.2 Canonical Huffman via Code Lengths: The Brilliant Compression

Instead of storing:

- Full tree structure (large), or
- Symbol-to-code mapping (variable size)

DEFLATE stores only:

- **Code length for each symbol** (4 bits each, 19 maximum length)
- **Canonical reconstruction rules**:
  1. Sort symbols by code length
  2. Assign first code of length k as 0
  3. Each subsequent code = previous code + 1
  4. When increasing length, left-shift and add zeros

Example

Canonical Huffman Example:
Given code lengths: A=2, B=2, C=3, D=3, E=3

1. Sort by length: A(2), B(2), C(3), D(3), E(3)
2. Length 2: Start with 00
3. Assign: A=00, B=01 (00+1)
4. Length 3: Start with 100 (01¡¡1)
5. Assign: C=100, D=101, E=110

Decoder only needs code lengths to reconstruct this!

6.4.3 Run-Length Encoding of Code Lengths (19-Symbol Alphabet)

Code lengths themselves are compressed using RLE on a 19-symbol alphabet:

- 0-15: Actual code length 0-15
- 16: Repeat previous length 3-6 times (2 extra bits)
- 17: Repeat zero 3-10 times (3 extra bits)
- 18: Repeat zero 11-138 times (7 extra bits)

This compression means the Huffman tree description is itself heavily compressed!

6.4.4 Why Canonical Codes Enable O(1) Decoding

With canonical codes, decoding becomes a table lookup:

```
// Precomputed tables from code lengths
min_code[length] = smallest code of this length
max_code[length] = largest code of this length
first_symbol[length] = first symbol with this length

// Decoding loop
while (need more symbols) {
    code = peek_bits(MAX_BITS);  // Look at next bits
    for (len = 1 to MAX_BITS) {
        if (code <= max_code[len]) {
            symbol = first_symbol[len] + (code - min_code[len]);
            consume_bits(len);
            break;
        }
    }
}
```

Listing 1: Canonical Huffman Decoding (Pseudocode)

This is **O(1)** per symbol (constant-time lookup), crucial for fast decompression.

6.5 Block Structure and Streaming Format

DEFLATE supports streaming through independent compression blocks.

6.5.1 Three Block Types: Stored, Fixed Huffman, Dynamic Huffman

Type	Bits	Use Case
Stored	00	Uncompressible data
Fixed Huffman	01	Default trees (predefined)
Dynamic Huffman	10	Custom trees per block
Reserved	11	(Error)

- **Stored blocks**: Raw data with length header (for already-compressed data)
- **Fixed blocks**: Use predefined Huffman trees (no tree description overhead)
- **Dynamic blocks**: Custom optimized trees (best compression)

6.5.2 Bit-Level Layout: BFINAL, BTYPE, and Data

Each block has precise bit-level layout:

```
// Block header
BFINAL: 1 bit  (1 = last block in stream)
BTYPE:  2 bits (00, 01, or 10)

// If BTYPE == 00 (Stored block)
//     Skip to next byte boundary
//     LEN: 16 bits (block length)
//     NLEN: 16 bits (~LEN for error checking)
//     Data: LEN bytes of raw data

// If BTYPE == 01 (Fixed Huffman)
//     Compressed data using fixed trees

// If BTYPE == 10 (Dynamic Huffman)
//     HLIT: 5 bits (257-286 literal/length codes)
//     HDIST: 5 bits (1-30 distance codes)
//     HCLEN: 4 bits (4-19 code length codes)
//     Code lengths for code length alphabet (HCLEN+4 times)
//     Compressed literal/length and distance code lengths
//     Compressed data
```

Listing 2: DEFLATE Block Structure

6.5.3 Streaming Support: Independent Blocks and Reset Points

- Each block compressed independently
- History buffer **does not reset** between blocks (maintains 32KB window)
- Allows streaming compression of unlimited data
- Enables random access in some formats (ZIP seeks to file boundaries)

6.6 Step-by-Step Compression Walkthrough

Let's trace through compressing a simple English phrase.

6.6.1 Input: "The cat sat on the mat"

Assume ASCII encoding (simplified for example):

```
Position: 01234567890123456789012345 6
Text:     The cat sat on the mat
Indices:  0         1         2
```

6.6.2 LZSS Processing: Finding Matches

1. Position 0-2: ”The” - No match in empty history → Output literals ’T’,’h’,’e’
2. Position 3: ’ ’ (space) - No match → Output literal ’ ’
3. Position 4-6: ”cat” - No match → Output ’c’,’a’,’t’
4. Continue... Position 18-20: ”the” matches position 0-2!
5. Match length = 3, distance = 18
6. Encode as (length=3, distance=18)

Final LZSS output (simplified):

```
Literals: T h e   c a t   s a t   o n   (length,distance=3,18) m a t
Symbols: 84 104 101 32 99 97 116 32 115 97 116 32 111 110 (257,18) 109 97 116
```

(Note: 257 is the symbol for length=3 in DEFLATE’s encoding)

6.6.3 Building Canonical Huffman Tables

From the symbol stream, compute frequencies:

Symbol	**Frequency**
32 (space)	4
97 (’a’)	3
116 (’t’)	3
Others	1 each
257 (length=3)	1
18 (distance)	1

Build Huffman trees, then extract code lengths. For example, space (32) gets shortest code.

6.6.4 Bitstream Assembly: Final Compressed Output

1. Write block header: BFINAL=1, BTYPE=10 (dynamic)
2. Write HLIT, HDIST, HCLEN values
3. Write compressed code lengths for the 19 code-length symbols
4. Write compressed literal/length and distance code lengths
5. Write compressed data using the Huffman codes
6. Pad final byte with zeros if needed

6.7 Compression Levels and Performance Trade-offs

zlib (the reference implementation) offers 9 compression levels.

6.7.1 zlib's 9 Levels: From "fast" to "best"

Level	Strategy	Use Case
0	No compression (store only)	Testing/disabled
1	Fastest LZ77, lazy matching off	Real-time compression
6	Default: moderate search, lazy matching	General purpose
9	Full search, optimal parsing	Archival storage

Levels mainly differ in:

- Hash table size and chain length
- Lazy matching on/off
- Match selection heuristics

6.7.2 Speed vs. Ratio: Why Level 6 is the Default

The default level 6 represents the "knee" in the speed/ratio curve:

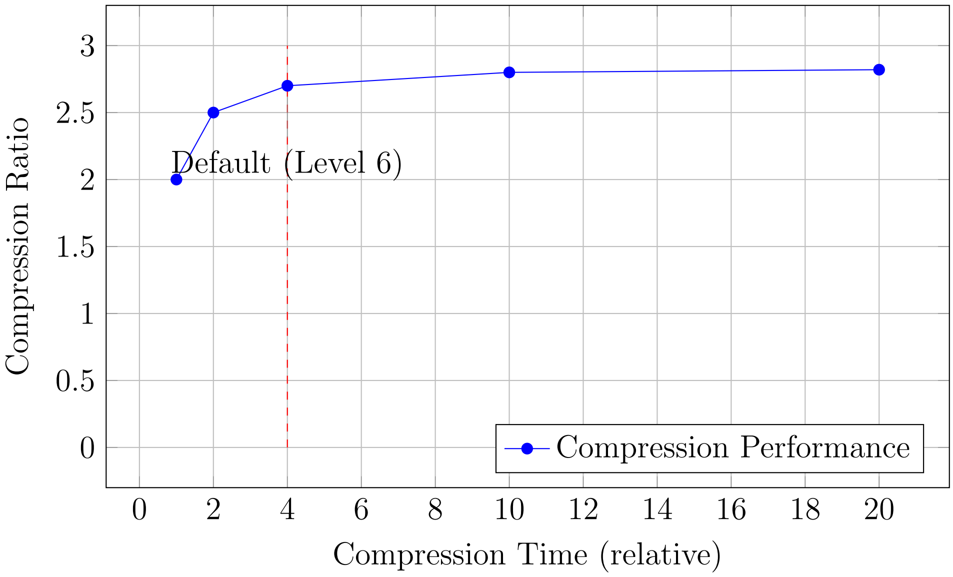


Figure 5: Trade-off between compression time and ratio in zlib

Level 9 is only ∼5% better than Level 6 but takes 2-5x longer.

6.7.3 Memory Footprint: 32KB to 256KB Windows

- **Default**: 32KB history buffer + 16KB hash table ≈ 48KB

- **Maximum**: Some implementations allow 256KB windows
- **Modern systems**: Typically use 32KB-64KB for cache efficiency

6.8 Practical Applications and File Formats

6.8.1 gzip Format: Headers, Footers, and CRC32

A gzip file contains:

```
+--------+-----+-----+----+-----+-----+------+------+
| Header | Extra| Name |Comment|HCRC | DEFLATE | CRC32 | ISIZE |
+--------+-----+-----+----+-----+-----+------+------+
  10 bytes optional optional 2    compressed  4      4
```

- **Header**: Magic bytes (0x1F, 0x8B), compression method (8=DEFLATE)
- **CRC32**: 32-bit checksum of uncompressed data
- **ISIZE**: Uncompressed size modulo 2^{32}

6.8.2 ZIP Format: Multiple Files and DEFLATE

ZIP is a container format supporting multiple compression methods:

- Each file compressed independently with DEFLATE
- Central directory at end enables random access
- Supports encryption, comments, extra fields

6.8.3 PNG: DEFLATE on Filtered Scanlines

PNG applies a **filter** to each scanline before DEFLATE:

- Subtract pixel from left pixel (Sub filter)
- Subtract pixel from above pixel (Up filter)
- Various predictors to create more compressible data
- Then DEFLATE compresses the filtered bytes

6.8.4 HTTP Compression: Accept-Encoding Header

Web servers compress responses on-the-fly:

```
Request:  GET /page.html HTTP/1.1
          Accept-Encoding: gzip, deflate

Response: HTTP/1.1 200 OK
          Content-Encoding: gzip
          [gzip-compressed body]
```

Saves 60-80% bandwidth for text content (HTML, CSS, JS).

6.9 Limitations and When DEFLATE Fails

6.9.1 Poor Performance on Already-Compressed or Random Data

- **Already compressed**: JPEG, MP3, ZIP files → DEFLATE can't compress further
- **Encrypted data**: Random-looking → No patterns to find
- **Result**: May actually *increase* size due to overhead

Good compressors detect this and use ”stored” (uncompressed) blocks.

6.9.2 The ”Shakespeare Problem”: Global vs. Local Repetition

DEFLATE's 32KB window can't capture **global repetition**:

Example

Shakespeare's Complete Works:

- Word ”the” appears thousands of times
- But only instances within 32KB of each other are linked
- Distant repetitions (across chapters) aren't captured
- Result: Compresses to $\sim$2 bits/byte instead of potential $\sim$1.5

Modern compressors (LZMA, zstd) use larger dictionaries (MBs) for this case.

6.9.3 Why Not Arithmetic Coding? Patents, Speed, and Complexity

In the 1990s, arithmetic coding was:

- **Patented**: IBM held key patents
- **Slower**: 2-10x slower than Huffman

- **Complex**: Harder to implement correctly
- **Bit-level operations**: Unfriendly to byte-oriented systems

Today, arithmetic coding is patent-free but DEFLATE's ecosystem is entrenched.

6.9.4 Security Issues: CRIME, BREACH, and Compression Side-Channels

Compression can leak information:

- **CRIME (2012)**: HTTPS compression reveals session cookies
- **BREACH (2013)**: HTTP compression reveals CSRF tokens
- **Principle**: If secret affects compressed size, attacker can deduce it
- **Solution**: Disable compression on sensitive data or use random padding

6.10 Modern Alternatives and Successors

6.10.1 LZMA and 7-Zip: Better Compression, Slower Speed

LZMA (Lempel-Ziv-Markov chain Algorithm):

- **Larger window**: Up to 4GB vs DEFLATE's 32KB
- **Range coding**: Arithmetic coding variant
- **Better compression**: 30-50% smaller than DEFLATE
- **Slower**: 5-10x slower compression
- **Used in**: 7z format, xz utilities, software installers

6.10.2 Zstandard (zstd): Facebook's Balanced Alternative

zstd (2016) aims to replace DEFLATE:

- **Faster**: 5-10x faster decompression than zlib
- **Better compression**: 10-20% better than zlib at same speed
- **Modern features**: Dictionary training, reproducible builds
- **Adoption**: Linux kernel, SquashFS, Facebook, Netflix

6.10.3 Brotli: Google's Web-Optimized Successor

Brotli (2013) targets web compression:

- **Static dictionary**: 120KB of common web strings
- **Better text compression**: 20-26% better than gzip
- **Slower compression**: But fast decompression
- **Adoption**: CDNs, web servers, browsers

6.10.4 Why DEFLATE Still Dominates: The Legacy Effect

Despite better alternatives, DEFLATE persists because:

- **Ubiquitous support**: Every system has zlib
- **Standardized**: RFC 1951, no licensing issues
- **Good enough**: For most use cases
- **Network effects**: Hard to change entrenched standards
- **Hardware support**: Some processors have DEFLATE acceleration

6.11 Hands-On Analysis and Debugging

6.11.1 Using gzip -v -l and infgen for Inspection

```
# Basic compression
gzip -9 file.txt              # Maximum compression
gzip -1 file.txt              # Fastest compression

# Analyze compressed file
gzip -l file.txt.gz           # List compression ratio
gzip -v -l file.txt.gz        # Verbose listing

# Decompress with verbose output
gzip -v -d file.txt.gz        # Show compression ratio

# Use infgen for detailed DEFLATE analysis
infgen -d file.gz             # Disassemble DEFLATE stream
```

6.11.2 Visualizing Matches and Huffman Trees

Tools for deeper analysis:

- **puff**: Educational DEFLATE implementation (readable C)
- **infgen**: DEFLATE stream disassembler
- **Custom scripts**: Parse zlib output with ZLIB_DEBUG

6.11.3 Benchmarking: Compression Ratio vs. Speed

```bash
# Time compression
time gzip -9 large_file.txt

# Compare compression levels
for level in {1..9}; do
    echo -n "Level $level: "
    gzip -c -$level file.txt | wc -c
done

# Benchmark suite
pigz                 # Parallel gzip implementation
zstd -b              # zstd benchmark mode
```

6.12 Summary and Key Insights

6.12.1 Why DEFLATE Dominated for 30+ Years

1. **Right trade-offs**: Fast decode over maximum compression
2. **Clever engineering**: Canonical Huffman, compact tree representation
3. **Open standard**: RFC 1951, no patents, reference implementation
4. **Good enough**: Adequate compression for most real-world data
5. **Ecosystem**: Tools, libraries, hardware support

6.12.2 Lessons for Compression Engineers: Practical Beats Perfect

- **Decode speed matters more than encode speed** for distribution formats
- **Implementation simplicity** leads to wider adoption
- **Open standards** defeat technically superior proprietary formats
- **Backward compatibility** is a powerful force
- Sometimes **”good enough” wins over ”best”**

6.12.3 Looking Forward: The Future of Compression Standards

- **DEFLATE will persist** in legacy systems for decades
- **zstd and Brotli** gaining ground in new applications
- **Hardware acceleration** becoming common (Intel QAT, AWS Nitro)
- **AI/ML approaches** showing promise but not yet practical
- The next "DEFLATE" will need to be **10x better** to displace it

Important

Final Thought: DEFLATE represents a peak in practical algorithm design. It shows how understanding both theory (information theory) and practice (hardware constraints, use cases) leads to enduring solutions. Study DEFLATE not just to use it, but to learn how to design algorithms that stand the test of time.

End of Chapter Exercises

Exercise 6.0

Exercise 6.1: DEFLATE Architecture Understanding
Explain in your own words why DEFLATE uses a two-stage approach (LZSS + Huffman) instead of just one of these techniques. Consider:

(a) What types of redundancy does LZSS exploit that Huffman cannot?

(b) What types of redundancy does Huffman exploit that LZSS cannot?

(c) Why can't Huffman coding alone achieve good compression on typical text data?

(d) Why can't LZSS alone achieve optimal compression?

Exercise 6.1

Exercise 6.2: LZSS Match Analysis
Given the input string: "abracadabraabracadabra"

(a) Using a sliding window of size 12, trace through LZSS processing and show the output as a sequence of literals and (length, distance) pairs.

(b) Calculate the compression ratio if literals are 8 bits and each (length, distance) pair requires 24 bits (8 for length, 16 for distance).

(c) What would be the output if the sliding window was only size 6 instead of

12?

Exercise 6.2

Exercise 6.3: Distance Encoding Practice
Using DEFLATE's distance encoding scheme (logarithmic ranges):

(a) Encode the following distances: 1, 5, 10, 100, 1000, 20000

(b) For each, specify: symbol number, base distance, extra bits required, and final binary encoding.

(c) What is the maximum distance that can be encoded without extra bits?

(d) Why does DEFLATE use logarithmic encoding instead of direct binary representation?

Hint: Refer to RFC 1951 Section 3.2.5 for the exact distance code table.

Exercise 6.3

Exercise 6.4: Canonical Huffman Construction
Given the following symbol frequencies for a small alphabet:

Symbol	**Frequency**
A	40
B	30
C	15
D	10
E	5

(a) Build a Huffman tree and determine code lengths for each symbol.

(b) Generate the canonical Huffman codes from these lengths.

(c) Show the exact bit sequence DEFLATE would store to represent these code lengths.

(d) Calculate the average code length and compare it to the entropy.

Exercise 6.4

Exercise 6.5: DEFLATE Block Analysis
Consider the following DEFLATE block header (in binary):

1 10 01101 00101 0100

Where the bits represent: BFINAL (1), BTYPE (10), HLIT (5 bits), HDIST (5 bits), HCLEN (4 bits).

(a) Decode each field and explain what it means.

(b) What type of block is this?

(c) How many literal/length codes will follow?

(d) How many distance codes will follow?

(e) How many code-length codes will follow?

Exercise 6.5

Exercise 6.6: Compression Level Trade-offs
You are designing a compression system for:

(a) A web server compressing HTML pages in real-time

(b) An archival system storing historical documents

(c) A mobile app with limited CPU and battery

For each scenario:

- Recommend a DEFLATE compression level (0-9) with justification
- Explain whether you would use fixed or dynamic Huffman blocks
- Discuss any modifications you might make to the standard DEFLATE parameters

Exercise 6.6

Exercise 6.7: DEFLATE vs. Modern Alternatives
Compare DEFLATE with two modern alternatives (choose from LZMA, zstd, or Brotli):

(a) Create a comparison table covering: compression ratio, speed (encode/decode), memory usage, and typical use cases.

(b) For each of the following, recommend which compressor to use and why:

  - Compressing software updates for download
  - Real-time compression in a database
  - Web assets on a content delivery network

(c) Explain why DEFLATE is still widely used despite newer, better algorithms.

Exercise 6.7

Exercise 6.8: Implementation Challenge
Write pseudocode for a DEFLATE decoder that:

(a) Reads the block header and identifies block type

(b) For dynamic blocks, reconstructs the Huffman trees from code lengths

(c) Decodes the compressed data using canonical Huffman decoding

(d) Handles both literals and (length, distance) pairs

Focus on the key algorithms, not edge cases or optimizations. Your pseudocode should show the overall structure and critical steps.

Exercise 6.8

Exercise 6.9: Security Analysis
The CRIME and BREACH attacks exploit compression side-channels:

(a) Explain the fundamental vulnerability that enables these attacks.

(b) Why does DEFLATE's LZSS stage make this vulnerability particularly dangerous?

(c) Propose three different strategies to mitigate this risk while still using compression.

(d) Would switching to a different compression algorithm (like zstd or Brotli) solve the problem? Why or why not?

Exercise 6.9

Exercise 6.10: Research and Reflection

(a) DEFLATE was standardized in 1996. Identify three technological constraints from that era that influenced DEFLATE's design decisions.

(b) If you were designing DEFLATE today, what would you change given modern hardware (multi-core CPUs, gigabytes of RAM, SSD storage)?

(c) Research the current state of hardware-accelerated compression (Intel QAT, AWS Nitro, etc.). How might these technologies change the optimal compression algorithm design?

(d) Predict: Will DEFLATE still be widely used in 2030? Justify your answer.

Important

Exercise Guidelines:

- **Exercises 6.1-6.5**: Core concepts – recommended for all students
- **Exercises 6.6-6.8**: Intermediate application – for deeper understanding
- **Exercises 6.9-6.10**: Advanced topics – for interested students or projects
- **Time estimates**: Basic exercises (30 min each), advanced (60-90 min each)
- **Resources**: Refer to RFC 1951, zlib source code, and infgen tool for hands-on exploration

Optional Project Challenge

Build a Simple DEFLATE Analyzer: Write a program in your preferred language that:

- Takes a gzip file as input
- Parses the gzip header and trailer
- Extracts the DEFLATE stream
- Identifies block boundaries and types
- Reports basic statistics (compression ratio, block types used, etc.)
- **Bonus**: Visualize the Huffman trees or match distances

Use existing libraries for decompression, but implement the analysis logic yourself. This project will deepen your understanding of the actual byte-level format.